

Curso de Windows Presentation Foundation

WPF

INDICE

BIENVENIDO AL CURSO DE WINDOWS PRESENTATION FOUNDATION.....	6
DE QUE VA ESTE CURSO.....	7
Los antecedentes	7
¿A quienes va dirigido?	8
¿Cómo está organizado?	8
Capítulo I INTRODUCCIÓN	11
Lección 1 Las bases de WPF y XAML.....	11
1.1 El Modelo de las Aplicaciones de Windows Vista	11
1.2 XAML.....	13
1.3 XAML y el code-behind	14
1.4 Ventajas de XAML y WPF	18
1.5 El Modelo Contenedor Contenido de XAML.....	20
1.6 Atributos	22
Lección 2 Construya su primera aplicación con WPF.....	23
2.1 Instalación de WPF	23
2.2 Crear una aplicación WPF para ejecutar como aplicación de escritorio.....	24
2.3 Crear una aplicación WPF para ejecutar en un navegador	27
2.4 Crear una aplicación WPF híbrida (Windows + Página)	30
Capítulo II CONCEPTOS BÁSICOS	32
Lección 3 Layout	32
3.1 StackPanel.....	33
3.2 DockPanel	34
3.3 Grid	36
3.4 WrapPanel	42
3.5 Canvas.....	44
Lección 4 Controles	46
4.1 Botones	48
4.2 Slider y Scroll.....	53
4.3 Controles de Texto.....	54
4.4 Etiquetas	56
4.5 Selectores	58
4.6 Menús	61
4.6 ToolBars	63
4.7 Expander	64
Lección 5 Eventos y Comandos	65

5.1 Enrutamiento de los eventos.....	66
5.2 Comandos en controles WPF	70
Lección 6 Enlace a Datos.....	79
6.1 Binding	79
6.2 Fuentes de datos	81
6.3 Colecciones como fuentes de datos	97
6.4 De cuando fuente y destino no concuerdan.....	100
Capítulo III GRÁFICOS	110
Lección 7 Figuras.....	110
7.1 Figuras básicas	110
7.2 Decoradores.....	117
7.3 Figuras vs Controles	122
7.4 Figuras Avanzadas.....	123
Lección 8 Brochas.....	131
8.1 Brochas de Colores enteros	131
8.2 Gradientes de colores.....	133
8.3 Tapizando tu fondo (TileBrush).....	143
8.4 Pinceles	164
Lección 9 Efectos Visuales	169
9.1 Efectos de transparencia	169
9.2 Efectos de Bitmap	176
Lección 10 Transformaciones.....	195
10.1 TranslateTransform	197
10.2 RotateTransform.....	199
10.3 ScaleTransform	201
10.4 SkewTransform	205
10.5 MatrixTransform	207
10.6 TransformGroup	209
Capítulo IV ESTILOS Y PLANTILLAS.....	212
Lección 11 Estilos	212
11.1 Personalizando sin estilos	212
11.2 Personalizando con estilos.....	215
11.3 Definición explícita de estilos	218
11.4 Estilos basados en otros estilos	220
Lección 12 Plantillas de Datos.....	222
12.1 Plantilla de datos para aplicar a un contenido simple	225
12.2 Plantilla para aplicar a un contenido no simple.....	227
12.3 Plantillas a Colecciones de Elementos	230
Lección 13 Triggers.....	244
13.1 Property Triggers	245
13.2 DataTriggers.....	248
13.3 EventTriggers	253
Lección 14 Plantillas de Controles.....	256

14.1 Plantillas visuales	257
14.2 Plantillas de controles compuestos	267
Capítulo V ELEMENTOS AVANZADOS	277
Lección 15 Animaciones	277
15.1 ¿Qué entendemos por animación?	277
15.2 Tipos de animaciones	283
15.3 Animaciones From/To/By (haciendo un pase raso del balón).....	284
15.4 Animaciones Key Frame (rebotando el balón)	287
15.5 AnimationPath (el pase de un crack)	294
Lección 16 Media	299
16.1 Media Element.....	299
16.2 Operaciones de Reproducción.....	301
16.3 Animación con Audio y Video	303
Lección 17 Gráficos 3D	307
17.1 Combinando los dos mundos	307
17.2 Viewport3D y ModelVisual3D.....	309
17.3 Primitivas	310
17.4 Mallas.....	311
17.5 Cámaras	316
17.6 Luces	319
17.7 Materiales.....	323
17.8 Transformaciones en 3D	326
17.9 Detección de Colisión (Hit Testing)	327
Lección 18 Documentos.....	333
18.1 Documentos Fluidos	333
18.2 Documentos Fijos	354
18.3 Visualizando Documentos	355
18.4 Anotaciones	359
18.5 Paquetes y XPS.....	364

BIENVENIDO AL CURSO DE WINDOWS PRESENTATION FOUNDATION

En este curso usted verá como crear sus primeras aplicaciones usando Windows Presentation Foundation (WPF). Le enseñaremos a utilizar los nuevos recursos de presentación de esta nueva plataforma de Microsoft con los que podrá elevar a niveles excitantes la calidad y funcionalidad de las interfaces de usuario de sus aplicaciones. Lo introduciremos en la nueva filosofía de programación declarativa con el lenguaje XAML que ayuda a separar la interfaz de presentación del código .NET en el que programe la lógica del negocio de su aplicación. Con WPF verá facilitados y enriquecidos conocidos conceptos como controles, eventos, enlace a datos, así como podrá incorporar nuevos elementos como transformaciones, plantillas, estilos, animaciones, media, 3D, documentos.

El curso está dividido en 5 capítulos, y cada capítulo en lecciones. Los capítulos y lecciones vienen acompañados de los proyectos con el código de los ejemplos, así como de videos ilustrativos que reforzarán el contenido de los temas tratados y lo ayudarán a experimentar por su cuenta.

Le recomendamos también que vea la aplicación integral Visor de Cursos MSDN que hemos desarrollado íntegramente con WPF y de la cual usted puede descargar completamente el código fuente. Con esta aplicación usted puede visualizar e interactuar con el curso.

¡Deseamos que disfrute este nuevo y emocionante universo que es WPF! Esperamos por sus comentarios y observaciones.

DE QUE VA ESTE CURSO

Los antecedentes

Cuando en 1985 la primera versión de Windows salió al mercado, y aún durante algún tiempo después, los programadores por lo general escribían aplicaciones monolíticas y aisladas, rústicas y aburridas. Windows abrió el camino al uso y desarrollo de componentes de software, y a los mecanismos de composición de esas componentes para desarrollar aplicaciones, dando soporte a una concepción interactiva y enriquecedora de usar las aplicaciones a través de interfaces gráficas de usuario, no en balde Windows se llamó Windows por ser el concepto de ventana gráfica la novedad fundamental. El gran salto cualitativo de un sistema que se nombraba por cómo trabajaba (**DOS**) a un sistema que debía su nombre a cómo tú trabajabas con él (**Windows**).

Pero en los 90s, posiblemente junto con la adolescencia o juventud de muchos de los lectores, llegó la WWW. Los ordenadores se estaban conectando más y más hasta formar el entramado que terminó llamándose Web y surgió el concepto de navegar en la Web. Las primeras aplicaciones para navegar (browser) se limitaban simplemente a desplegar visualmente contenidos en el ordenador de escritorio del usuario, imitando el efecto de algo así como repasar las páginas de un periódico. Estos “contenidos” estaban descritos en un lenguaje de marcas HTML, que para cordura colectiva fue aceptado por todos.

Y junto a la Web fueron apareciendo otras posibilidades. Internet Explorer 4, por ejemplo, le añadió guiones (scripts). Los guiones son interpretados por los navegadores y permitían dotar de un comportamiento y una personalización a los contenidos que hasta ahora solo se desplegaban pasivos. Ahora con los guiones los contenidos HTML podían reaccionar a eventos interactuando con los usuarios y permitiendo a los navegadores ejecutar alguna lógica y reaccionar más rápidamente sin tener que estar viajando al servidor cada vez que se hacía una mínima interacción.

No obstante los avances en la Web y en los navegadores, las aplicaciones de escritorio desarrolladas sobre Windows seguían teniendo por lo general una interfaz más excitante y un comportamiento más rico, complejo y rápido que el de la mayoría de las aplicaciones Web. Por otra parte programar y depurar aplicaciones Web seguía siendo mucho más difícil que las aplicaciones Windows.

Pero también es cierto que la forma de trabajar con la Web introdujo concepciones que no se tenían en las aplicaciones de escritorio Windows, como la forma simple de instalar una aplicación y la navegación hacia detrás y hacia delante basada en páginas.

Y en eso con el nuevo milenio llegó Microsoft .NET. La más moderna y evolucionada plataforma de desarrollo de software que permitió un rápido desarrollo de las aplicaciones

tanto de escritorio como Web, usando muchas más y mejores componentes de software, basándose en estándares aceptados de facto como un modelo orientado a objetos seguro en sus tipos (type safe), http, XML y el hacer a la Web asequible programáticamente a través de los servicios Web. Y por demás todo esto en un escenario agnóstico en cuanto al lenguaje de programación si éste satisface un conjunto de requerimientos comunes conocido como CLS (Common Language Specification).

Durante su aún joven existencia .NET ha demostrado con creces sus bondades e incrementando exponencialmente la cantidad y versatilidad de las aplicaciones, y lo que es muy importante: la productividad de los desarrolladores de software. Sin embargo, mantener una sistemática renovación, para no dormirse en el éxito de sus productos, ha sido una constante en Microsoft que ha jalónado el desarrollo. A pesar de la riqueza en capacidades de desarrollo introducidas por .NET, tanto para el mundo Windows como para el mundo Web, lo cierto es que aún existe una brecha en la forma de desarrollar aplicaciones y la de integrar las bondades de ambos mundos. **WPF** (Windows Presentation Foundation) es el nuevo producto de Microsoft para salvar esta brecha. WPF parece ser uno de los más excitantes desarrollos para la programación en Windows. Sobre cómo desarrollar aplicaciones en WPF tratará este curso.

¿A quienes va dirigido?

Comenzar un curso sobre un nuevo tema pretendiendo ser “independiente” de lo que Ud. conoce es caer en la vieja trampa del **qué pongo** y el **qué supongo**. De modo que partimos de que a estas alturas, si es que no estaba en hibernación, Ud. estará familiarizado con Windows y con su espectacular aporte al desarrollo de interfaces gráficas de usuario para las aplicaciones desarrolladas sobre éste.

Este no es un curso para aprender a programar, por tanto tener experiencia en desarrollo orientado a objetos con algún lenguaje de programación de .NET (preferiblemente C#), y que Ud. haya al menos romanceado con una herramienta de desarrollo como Visual Studio, son requerimientos para pasar este curso. Y por supuesto que Ud. debe ser al menos un usuario de la Web por lo que HTML y XML no le deben resultar desconocidos.

Pero si aún no se ha introducido en .NET no se aflijta, puede remitirse a los cursos existentes los cuales puede acceder en <http://www.microsoft.com/spanish/msdn/spain/cursosonline.mspx>

¿Cómo está organizado?

Aunque un buen desarrollador de WPF debería tener una sólida base escribiendo aplicaciones WPF completamente en código de un lenguaje .NET, abordar este curso con este enfoque lo haría muy extenso y competiría contra nuestro objetivo fundamental y es que Ud. pueda poner rápidamente manos a lo obra y pueda empezar a hacer sus primeras aplicaciones sobre WPF. Es por ello que el curso se basa en la combinación **lenguaje de marcas declarativo + código procedural .NET**. La descripción de las interfaces visuales de

interacción de las aplicaciones serán presentadas en un nuevo lenguaje de marcas **XAML** (eXtensible Application Markup Language) y el código que implementa la lógica de la aplicación asequible a través de esta interfaz será desarrollado en el lenguaje orientado a objetos **C#**.

Por la imposibilidad de abordar en un solo curso la basta amplitud de recursos y especificidades de WPF y XAML, este curso tratará aquellos temas más relevantes ilustrándolos con ejemplos, listados de código e imágenes con el despliegue de los resultados. Todos los códigos de los ejemplos podrán ser descargados del sitio del curso. El curso culminará con el desarrollo de una aplicación general que ilustra el uso de la mayoría de los temas tratados.

El curso está dividido en Capítulos y cada Capítulo consta de varias lecciones cada una sobre un tema particular. Las lecciones vienen acompañadas de un video guiado que ilustra cómo implementar los ejemplos tratados y variaciones de estos.

Si Ud. se encuentra por primera vez con WPF es recomendable que siga el curso de manera secuencial en el orden en que están las lecciones. Ud. siempre podrá saltar hacia detrás y hacia delante si así lo desea a través de los enlaces.

El primer Capítulo es una introducción general a WPF. Con esta introducción se pretende ilustrar al lector en los conceptos más básicos que están detrás de WPF y XAML. Se presentan los primeros ejemplos reales y se explican las tres formas de crear aplicaciones con WPF: aplicaciones Windows para escritorio, aplicaciones para Web y una forma híbrida que mezcla bondades de Windows y bondades de Web

El Capítulo 2 está dedicado al trabajo con controles. En éste se trata el concepto de panel y cómo se distribuyen los elementos dentro del panel. Los paneles constituyen la pieza fundamental para organizar la apariencia visual de la aplicación. También se presentan aquí los controles básicos así como los diferentes eventos asociados a estos controles básicos y los nuevos mecanismos de propagación de eventos que introduce WPF. Finalmente la Lección Enlace a Datos nos muestra cómo se puede hacer el enlace con datos para llenar y personalizar los controles con información.

El Capítulo 3 está dedicado a Gráficos. Se comienza con las figuras y los recursos básicos para hacer figuras. Luego se tratan las "brochas y pinceles". También se tratan los efectos que se pueden hacer sobre las imágenes y las transformaciones gráficas a las que puede someterse una interfaz. WPF basa la visualización de todos sus elementos en la tecnología gráfica Microsoft DirectX de modo que los amantes de las aplicaciones gráficas tienen ahora facilidades para tocar el paraíso con sus dedos.

El Capítulo 4 está dedicado a estilos y plantillas. Los estilos nos permiten personalizar la apariencia de la interfaz de usuario de modo similar y mas amplio que lo que por ejemplo Ud. ahora puede hacer con los estilos en Microsoft Word. Las plantillas (*templates*) son un

recurso que nos facilitará expresar patrones de código y que propiciará la reutilización aumentando la productividad y flexibilidad.

El Capítulo 5 trata sobre temas más avanzados. Aunque imposible cubrirlos todos, y cubrir cada uno en profundidad, se trata aquí sobre cómo incluir en sus aplicaciones WPF, video, gráfica 3D, animación y documentos, algo que antes de WPF había que lograr con un trabajo bizarro y una amalgama de tecnologías.

El curso culmina con la exposición de una aplicación general que servirá a su vez de "visor del propio curso". Muchos de sus elementos participantes se habrán ido presentando en las lecciones del curso.

Capítulo I INTRODUCCIÓN

Windows Presentation Foundation (WPF) es una de las novedosas tecnologías de Microsoft y uno de los pilares de Windows Vista. WPF potencia las capacidades de desarrollo de interfaces de interacción integrando y ampliando las mejores características de las aplicaciones windows y de las aplicaciones web. WPF ofrece una amplia infraestructura y potencialidad gráfica con la que se podrán desarrollar aplicaciones de excitante y atractiva apariencia, con mayores y más funcionales facilidades de interacción que incluyen animación, vídeo, audio, documentos, navegación, gráfica 3D. WPF separa, con el lenguaje declarativo XAML y los lenguajes de programación de .NET, la interfaz de interacción de la lógica del negocio, propiciando una arquitectura Modelo Vista Controlador para el desarrollo de las aplicaciones.

Este primer capítulo es una introducción general a WPF. Con esta introducción se pretende ilustrar al lector en los conceptos más básicos que están detrás de WPF como el modelo de árbol de elementos de presentación y el lenguaje declarativo de marcas XAML. Se presentan los primeros ejemplos reales y se explican las formas de crear aplicaciones con WPF: aplicaciones Windows para escritorio, aplicaciones para Web y una forma híbrida que mezcla bondades de Windows y bondades de Web.

Lección 1 Las bases de WPF y XAML

1.1 El Modelo de las Aplicaciones de Windows Vista

Las aplicaciones de escritorio en Windows tienen sus fortalezas sobre las aplicaciones Web. Seguramente muchos de Uds. aún pueden sentir que la interactividad y estímulo sensorial de algunas aplicaciones Web parecen rústicas y poco atractivas en comparación con las que se logran en una aplicación de escritorio Windows. Programando aplicaciones de escritorio sobre .NET Ud. dispone de un sinfín de componentes y de medios para integrar la lógica de sus aplicaciones con una rica interfaz gráfica que a su vez saca ventaja del hardware del cliente en comparación con la neutralidad que deben mantener los navegadores Web (compare por ejemplo la rica apariencia y funcionalidad de Outlook con la simpleza e incomodidad de algunos Web Mail).

Los viajes de ida y vuelta entre el cliente y el servidor de las aplicaciones Web pueden atentar contra el rendimiento. Esto ha provocado que los controles Web sean más primitivos en comparación con los de Windows lo que induce a su vez a que los diseñadores de las

aplicaciones Web incorporen poca interactividad y efectos en sus aplicaciones para compensar la lentitud (que a pesar de los aumentos en los anchos de banda no compite aún contra el bus de datos de la placa madre). ¿Ud. no se ha molestado alguna vez cuando para ver el precio de su reserva de avión tiene que navegar a la página siguiente para que cuando compruebe que el precio no le viene bien y quiere probar con otro vuelo tenga que volver a la página anterior?

Sin embargo, las aplicaciones tradicionales de escritorio sobre Windows también tienen sus debilidades en comparación con las aplicaciones Web. Las aplicaciones Windows por lo general requieren ser instaladas, tarea que además se dificulta con el estar pendiente de las actualizaciones. Además es cierto que el paradigma de aplicaciones orientadas a páginas, navegando de una página a otra, así como el llevar el historial de navegación, se puede echar de menos en las aplicaciones Windows (a menos que claro Ud. logre un efecto similar “programando a cincel y martillo”).

Por otro lado a pesar de su riqueza interactiva las aplicaciones Windows tienen sus limitaciones para el despliegue de documentos y la integración de textos, imágenes, audio y video (¿ha intentado hacer algo de esto con las Windows Forms?). Lo más probable es que algunas de las aplicaciones que Ud. habrá apreciado con riqueza de medios y controles hayan tenido que desarrollarse bajo el embrollo de mezclar variadas tecnologías (controles Active X, Flash, PDF, etc) y con varios desarrolladores intentando con dificultad integrar sus trabajos y resultados.

El modelo de aplicación de Windows Vista se basa en las mejores características y experiencias de ambos mundos para integrarlos en un modelo unificado de programación basado a su vez en el código administrado de .NET. Un elemento integrante de ese modelo es WPF y es de WPF de los que trata este curso.

Hay dos componentes fundamentales en el modelo de aplicaciones de Windows Vista. Una es **WCF (Windows Communications Foundation)**, conocida por su nombre interno **Indigo**. Este es un sistema de comunicación basado en mensajes sobre múltiples transportes y a través de sistemas heterogéneos. La otra componente es **WPF (Windows Presentation Foundation)**, conocida por su nombre interno **Avalon**. WPF da soporte al desarrollo integrado de una rica, versátil, excitante, interactiva y personalizada capa de presentación de nuestras aplicaciones con la lógica del negocio del tema de la aplicación y todo ello ejecutando en un contexto confiable, seguro y productivo como es .NET.

¿No le ha ocurrido que sus clientes, e incluso Ud. mismo, se sienten insatisfechos por la pobre y poco personalizada apariencia visual de su aplicación? ¿No se ha visto disminuido el valor de la experticia que logró poner en la lógica de su aplicación porque la aplicación tiene una pobre, sobrecargada o poco estimulante interacción gráfica? ¿No ha querido Ud. que su aplicación de escritorio basada en formularios Windows tenga efectos y apariencias visuales como las que puede lograr por ejemplo con sus presentaciones en Power Point? o a

la inversa, ¿no ha querido Ud. que una presentación Power Point pueda tener más funcionalidad y reaccionar y mostrar datos acorde con la lógica de su negocio? Ahora con WPF podrá lograr que sus aplicaciones tengan una mezcla de ambas bondades.

1.2 XAML

XAML siglas de eXtensible Application Markup Language es el lenguaje declarativo propuesto por Microsoft para definir las interfaces de usuario de las aplicaciones. XAML se basa en una sintaxis bien formada en XML y su fácil extensibilidad. XAML propicia separar la definición de las interfaces de usuario de la lógica propia de la aplicación. En este sentido ofrece soporte para expresar el desarrollo de aplicaciones sobre la arquitectura conocida como **MVC** (Model View Controller) Modelo Vista Controlador.

La intención con XAML es que en un lenguaje declarativo se puedan definir los elementos que compondrán una interfaz de usuario para que estos puedan ser desplegados y conectados con la lógica de la aplicación mediante un motor de presentación que ejecuta sobre .NET Framework conocido como WPF. Algo así como que XAML es el pentagrama y la simbología para expresar la partitura pero WPF es la orquesta para ejecutarla!

XAML sigue las reglas sintácticas de XML. Cada elemento XAML tiene por tanto un nombre y puede tener uno o más atributos. Realmente XAML ha sido diseñado para establecer una correspondencia lo más directa posible con el CLR de .NET. Por lo general todo elemento en XAML se corresponde con una clase del CLR de .NET, en particular de WPF, y todo atributo XAML se corresponde con el nombre de una propiedad o de un evento de dicha clase.

A la inversa, aunque no todas las clases del CLR están representadas en XAML, aquellas clases que tengan un constructor por defecto y propiedades públicas pueden ser instanciadas declarativamente en código escrito en XAML.

Todos los nombres de elementos y de atributos XAML son sensitivos a mayúscula y minúscula. Los valores de los atributos, con independencia de su tipo deben escribirse entre comillas dobles ("").

La mayoría de los elementos XAML tienen la característica de que se despliegan visualmente (*render*), pueden recibir entrada de teclado y ratón, disparan eventos y saben visualizar el tamaño y posición de sus elementos contenidos (*hijos*).

La mayoría de los elementos XAML son elementos de interfaz de usuario y derivan de System.Windows.UIElement

Note en el código XAML del Listado 1- 1 el pedazo que nos define un botón (destacado en negrita). Al botón se le especifican los atributos ancho, alto, tipo de fuente, tamaño de la fuente, color de fondo y color de frente (por simplicidad hemos dejado que otros muchos atributos del botón tomen su valor predeterminado), el botón tiene un

contenido que en este caso es el texto Púlsame que aparece entre los dos tags <Button> y </Button>. La Figura 1 - 1 nos muestra cómo se vería ese botón al ser desplegado por WPF.

Por simplicidad, en muchos de los ejemplos de este curso hemos dejado que algunos atributos tomen su valor predeterminado. Sin embargo, hasta ahora los valores predeterminados de muchos atributos han ido cambiando de una versión de WPF a otra. Para el desarrollo de aplicaciones reales recomendamos poner explícitamente los valores que quiera garantizar para los atributos aún cuando estos coincidan con el valor predeterminado.

Listado 1- 1 Código XAML de ventana con un botón

```
<Window x:Class="WPFCursoOnline.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Ventana con botón" >
    <Button Width="200" Height="100" FontFamily="Consolas"
        FontSize="25" Background ="Blue" Foreground="White">
        Púlsame
    </Button>
</Window>
```

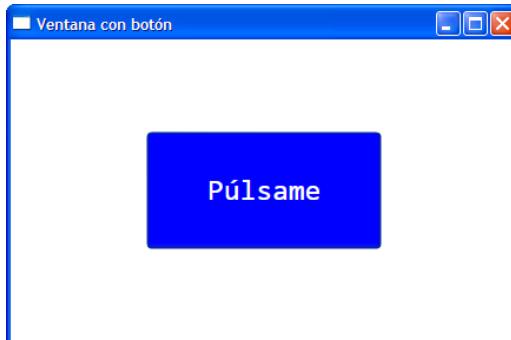


Figura 1 - 1 Ventana con botón sin acción asociada

1.3 XAML y el code-behind

Si hacemos clic sobre el botón de la aplicación de la Figura 1 - 1 no ocurrirá nada. Si queremos asociar una acción al evento de hacer clic sobre el botón debemos dar como valor del atributo Clic el nombre de un manejador para ese evento (tal y como se muestra en el Listado 1- 2).

En este caso queremos que al hacer clic el botón cambie de color. ¿Dónde definimos el manejador con la acción para lograr el cambio de color al pulsar el botón? Como XAML es un lenguaje de marcas declarativo, esta acción hay que describirla en un lenguaje procedural de .NET (C# en este curso). Esto es lo que se conoce como **code-behind** (o código en el trasfondo). En este caso el manejador, que hemos llamado miBoton_Clic, lo que hace es

cambiar en C# el valor de la propiedad Background. Pero para ello el manejador debe tener una forma de poder referirse al botón. Para lograr esto en el código XAML del Listado 2 se le da un nombre al botón al asociar al atributo Name el valor "miBoton". El código C# del manejador miBoton_Clic se muestra en el Listado 1-3

Listado 1- 2 XAML para botón con manejador de evento

```
<Window x:Class="WPFCursoOnline.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Botón que cambia de color con codenbehind" >
    <Button      Name="miBoton"      Clic="miBoton_Clic"
FontFamily="Consolas"
    Width="200" Height="100" Background ="Blue"
    Foreground="White" FontSize="25">
        Púlsame
    </Button>
</Window>
```

Listado 1- 3 Code-behind para el XAML del Listado 1-2

```
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace WPFCursoOnline
{
    public partial class Window1 : System.Windows.Window{
        bool azul=true;
        public Window1()
        {
            InitializeComponent();
        }

        //Manejador del evento Clic
        void miBoton_Clic(object sender, RoutedEventArgs e){
            if (azul)
                miBoton.Background = Brushes.Red;
            else
                miBoton.Background = Brushes.Blue;
            azul=!azul;
        }
    }
}
```

Si ahora se ejecuta la aplicación y se hace clic sobre el botón éste cambiará de color como se muestra en la Figura 1 - 2.

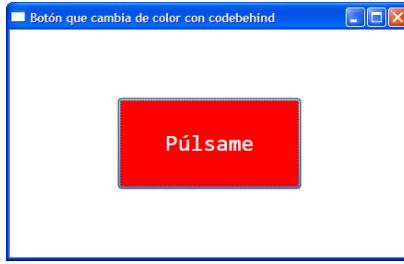


Figura 1 - 2 Botón con manejador de evento

El código C# del Listado 1- 3 puede ponerse **inline** dentro del código XAML como se muestra en negrita en el Listado 1- 4. Sin embargo, esto sería una práctica contraria a lo que quiere promoverse con el MVC para separar la capa de presentación de la capa de la lógica de la aplicación.

Listado 1- 4 XAML con código C# inline para el manejador del botón

```
<Window x:Class="WPFCursoOnline.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Botón con código C# inline para el manejador" >
    <Button Name="miBoton" Clic="miBoton_Clic" FontFamily="Consolas"
FontSize="25">
        Width="200" Height="100" Background = "Blue" Foreground="White"
        Púlsame
    </Button>
    <x:Code>
        <![CDATA[
            bool azul=true;
            void miBoton_Clic(object sender, RoutedEventArgs e){
                if (azul)
                    miBoton.Background = Brushes.Red;
                else
                    miBoton.Background = Brushes.Blue;
                azul=!azul;
            }
        ]]>
    </x:Code>
</Window>
```

Lo adecuado es poner este código con la implementación del manejador del evento, y el resto de la lógica de la aplicación, en un fichero (con extensión *xaml.cs*) separado del fichero (con extensión *.xaml*) que contiene el código XAML. Visual Studio nos genera

automáticamente el fichero *xaml.cs* con el esqueleto de código C# tal y como se muestra en el Listado 1- 5. Ud. deberá rellenarlo con el código necesario para completar la definición de los manejadores.

Listado 1- 5 Esquema de código para completar el code-behind

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace WPFCursoOnline
{
    public partial class Window1 : System.Windows.Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

En principio esto es todo lo que se necesita saber para acoplar su interfaz de usuario XAML con su lógica de funcionamiento en C#. En el XAML hay que ponerle nombre a los elementos que necesitemos referir en el código C# y darle como valor el nombre de un manejador a los atributos eventos que queremos tratar. El resto es llenar el código C# que queda en el trasfondo.

Para los más curiosos

Por lo pronto pudiera saltar a la sección siguiente, pero si tiene curiosidad por comprender un poco lo que ocurre tras bambalinas en el cómo WPF hace este acople entonces continúe con esta sección.

Note la palabra clave *partial* junto al nombre de la clase *Window1*. Esto le indica al compilador que ésta es una clase parcial y que el código de la misma, rellenado por el programador, debe acoplarse con el código .NET que corresponde a la definición de la interfaz escrita en el XAML y que será quien con su ejecución hará el despliegue y accionar de la capa de presentación.

El concepto de clase parcial es un aporte importante del .NET Framework 2.0. Si Ud. se fija en el código del Listado 3, en el mismo usamos la variable miBoton pero no aparece la instanciación e inicialización de un objeto Button para asignar a dicha variable. Tampoco aparece la definición del método InitializeComponent. Ud. puede imaginar que lo que ocurre es un acople de la clase parcial del Listado 1- 3 con una clase parcial como la que se muestra en el Listado 1- 6.

Listado 1- 6 Supuesta clase parcial para el acople con el code-behind

```
namespace WPFCursoOnline
{
    public partial class Window1 : System.Windows.Window
    {
        Button miBoton = new Button();
        void InitializeComponent()
        {
            miBoton.Width = 200;
            miBoton.Heigth = 100;
            miBoton.FontSize = 25;
            miBoton.Background = Brushes.Blue;
            miBoton.Foreground = Brushes.White;
            ...
        }
    }
}
```

Realmente Visual Studio no genera una tal clase. El proceso que ocurre es mucho más complejo y su descripción se sale de los objetivos de este curso. Cuando se compila el proyecto que contiene el (los) fichero (s) con extensión .xaml un árbol por cada XAML queda “serializado” e incrustado en el ensamblado resultante de la compilación. En ejecución el método InitializeComponent lo que hace es “recorrer” este árbol, crear las instancias correspondientes a los elementos XAML y establecer las asociaciones con las variables, en el ejecutable correspondiente al código C#, que hayan sido nombradas a través del atributo Name en cada elemento XAML.

Por otra parte en XAML, al igual que en XML, el valor que se le puede asociar a un atributo se escribe siempre en formato de un string. Por eso Ud. escribe `FontSize="25"` o `Background ="Blue"`. Es toda una maquinaria de convertidores de WPF (maquinaria que Ud. puede extender) la que a partir de aquí termina haciendo algo como `miBoton.FontSize = 25;` o `miBoton.Background = Brushes.Blue;`

1.4 Ventajas de XAML y WPF

Si el lector ha desarrollado alguna aplicación con Windows Forms usando Visual Studio, podrá pensar que lo que se ha explicado en la sección anterior es lo mismo que

podemos hacer con el diseñador de Visual Studio cuando implementamos una aplicación Windows Forms y arrastramos un botón desde la caja de herramientas para luego retocarle las propiedades con el editor de propiedades. En este caso es el Diseñador de Visual Studio quien está por detrás generando el código procedural C# correspondiente. Entonces toda esta nueva parafernalia del WPF para qué ¿el mismo vino con otra botella? ¿No es más complicado escribir este código en XAML que trabajar visualmente con el Diseñador de Visual Studio y el Editor de Propiedades?

Bueno, nadie ha dicho que con WPF vamos a renunciar a usar una herramienta visual de diseño integrada a Visual Studio. Una herramienta básica integrada a Visual Studio, de nombre Cyder, está en preparación (y por ello en este curso no ha sido utilizada). Una tal herramienta de diseño generaría código declarativo en XAML (como el que ahora hemos escrito a mano) en lugar del código C#.

Lo que hace más valioso a este enfoque de WPF basado en XAML es que al quedar la interfaz de usuario especificada con un lenguaje de marcas y declarativo, como es XAML, la hace legible y más fácilmente "retocable" por los humanos y, lo que es más importante, fácilmente susceptible de ser analizada y procesada por herramientas. Esto ofrece a terceros la posibilidad de crear herramientas de diseño visual que soporten XAML (de hecho ya se están desarrollando algunas).

La separación entre código XAML para expresar la interfaz y apariencia, y código de un lenguaje .NET como C# para expresar la lógica de la aplicación, facilitará que los diseñadores con capacidades artísticas (posiblemente con experiencias en los lenguajes de marcas como HTML pero poco habilidosos para la programación) puedan integrar mejor y con mas efectividad su trabajo con los programadores concentrados en la lógica de la aplicación (que por lo general son muy rústicos a la hora de diseñar una atractiva apariencia). Con ello se facilitará poder modificar la interfaz visual sin tener que tocar la lógica de la aplicación (incluso sin ver código C#) y viceversa.

XAML es extensible, como su nombre lo indica, lo que significa que los desarrolladores pueden crear sus propios controles y elementos personalizados. Como XAML es por naturaleza una representación textual en XML de los objetos de WPF, puede ser extendido por los desarrolladores por las técnicas de la programación orientada a objetos. De este modo objetos concebidos para la lógica de la aplicación pudieran ser expuestos vía XAML para que sean extendidos por los diseñadores visuales.

Pero es más, aunque XAML no es un lenguaje de programación procedural, y no está orientado a escribir la lógica de la aplicación, tiene elementos de naturaleza funcional que inciden sobre la propia visualización de la interfaz, tal es el caso por ejemplo de los elementos Trigger y Animation que permitirán darle dinamismo a la interfaz visual sin tener que llegar a C#. Ver Lección **Animaciones** y Lección **Triggers**.

1.5 El Modelo Contenedor Contenido de XAML

El núcleo del modelo de presentación de WPF se basa en los elementos de presentación. En el ejemplo de esta sección tenemos un elemento Button que es un control con contenido visual y comportamiento (puede responder a eventos) y un elemento Window que es un contenedor. En este caso el contenido del botón es un texto y por eso se puso directamente la cadena Púlsame dentro de los tags Button. Esto se podría haber escrito también asociando el valor "Púlsame" al atributo Content como se muestra en el Listado 1- 7

Listado 1- 7 Uso del atributo Content

```
<Window x:Class="WPFCursoOnline.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Botón que cambia de color con code-behind" >
    <Button Name="miBoton" Clic="miBoton_Clic"
FontFamily="Consolas"
    Width="200" Height="100" Background = "Blue"
    Foreground="White" FontSize="25" Content="Púlsame">
    </Button>
</Window>
```

Lo interesante es que en WPF el contenido de un botón no tiene por qué ser un texto sino que puede ser cualquier otro elemento (prácticamente cualquier objeto .NET).

El Listado 1- 8 nos muestra cómo definir un botón que tiene como contenido una imagen, lo que al ser desplegado por WPF se obtiene la Figura 1 - 3.

Listado 1- 8 XAML de botón con imagen dentro

```
<Window x:Class="WPFCursoOnline.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Botón con imagen dentro" >
    <Button Width="200" Height="120" Background = "AliceBlue" >
        <Image Source="Windito_Vistoso.gif" Height="100"
Margin="5"/>
    </Button>
</Window>
```

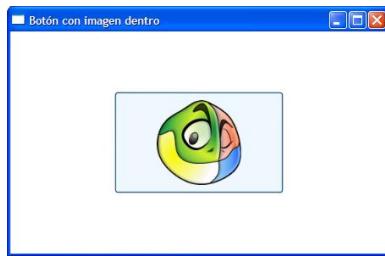


Figura 1 - 3 Botón con una imagen como contenido

La propia estructura sintáctica de XAML permite visualizar con facilidad la relación contenedor-contenido con cualquier nivel de anidamiento como se muestra en el Listado 1- 9

Listado 1- 9 Anidamiento de elementos en XAML

```
<Elemento N1="">
  <Elemento N11="">
    <Elemento 111="">
      ...
    </Elemento 111>
    <Elemento N112="">
      ...
    </Elemento N112>
  </Elemento N11>
  <Elemento N12="">
    ...
  </Elemento N12>
  ...
</Elemento N1>
```

Esta estructura sintáctica anidada corresponde a un árbol y como tal es entonces fácilmente analizable y recorrible programáticamente.

En este ejemplo Button, al igual que otras muchas clases de WPF como Label, ToolTip, CheckBox, que serán estudiadas en la Lección **Controles básicos**, derivan de la clase ContentControl. Un ContentControl solo puede contener un elemento, en este caso WPF puede desplegar el elemento contenido en dependencia del área que disponga el control. Esto no quiere decir que no se pueda lograr el efecto visual de que se despliegan más de un elemento dentro de un botón. Para contener más de un elemento se tiene un tipo de elemento llamado Panel. La distribución y forma en que serán desplegados los elementos definidos dentro de un panel se conoce como **Layout**. Los distintos tipos de paneles serán estudiados en la Lección **Layout**.

Listado 1- 10 XAML de botón que contiene imagen y texto

```
<Window x:Class="WPFCursoOnline.Window1"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```

Title="Botón con imagen y texto" >
<Button Width="220" Height="120" Background ="AliceBlue">
  <DockPanel>
    <Image Source="Windito Vistoso.gif" Height="100"
Margin="5"/>
    <TextBlock      FontFamily="Arial"      FontSize="20"
FontWeight="Bold"
      Foreground="Green">
      WINDITO
    </TextBlock>
  </DockPanel>
</Button>
</Window>

```

El Listado 10 muestra el uso del panel **DockPanel**. En este caso el contenido del botón será un panel que contiene una imagen (un elemento **Image**) seguido de un texto (un elemento **TextBlock**). El resultado se muestra en la Figura 1 - 4



Figura 1 - 4 Botón con imagen y texto

1.6 Atributos

Los atributos son la representación XAML de las propiedades de la clase correspondiente a un elemento.

Lamentablemente el término atributo también ha sido utilizado por .NET para nombrar al recurso de metainformación que se puede asociar a diferentes construcciones de los lenguajes .NET. Este uso del término no tiene nada que ver con uso que se le da en XAML.

Dependiendo del tipo de la entidad representada por el atributo, a estos pueden asignársele un valor de una de dos formas. Se le puede asignar un valor **inline**, como parte de la declaración del elemento o se le puede asignar un valor declarándolo explícitamente como un elemento anidado dentro del elemento que se está describiendo.

Cuando el atributo se le asigna valor inline, el valor en XAML se pone siempre como un string, es decir entre comillas (" "). Esto no quiere decir que el valor real correspondiente en su representación en el CLR sea un string. Por ejemplo cuando hacemos <Button

Width="20" .../> el string "20" corresponde al valor entero 20 en el CLR de .NET, cuando se hace <Button Background="AliceBlue" .../> el string realmente corresponde al objeto brocha Brushes.AliceBlue (a su vez una propiedad estática de tipo Brush y de nombre AliceBlue definida dentro de la clase Brushes. WPF se encarga de hacer las transformaciones a partir del string "AliceBlue" que es lo que se escribe originalmente en XAML. Esto podía ponerse en forma anidada explícitamente (Listado 1- 11).

Listado 1- 11 Atributo con valor declarado explícitamente

```
<Button Width = "100" Height = "30">  
  <Button.Background>  
    <SolidColorBrush Color = "AliceBlue" />  
  </Button.Background>  
</Button>
```

Lección 2 Construya su primera aplicación con WPF

2.1 Instalación de WPF

En esta lección le mostraremos cómo crear sus proyectos para WPF desde Visual Studio. Veremos como crear tres tipos de aplicaciones. Aplicaciones WPF que ejecutarán en Windows. Aplicaciones WPF Web y que ejecutarán dentro de un navegador (Internet Explorer) y en las que podrá manejar el concepto de página. Y también se verá como crear una aplicación Windows pero con páginas de modo de poder navegar entre dichas páginas.

Aplicable a:

Visual Studio 2005
Visual Studio 2008

Para practicar los temas de este curso Ud. necesita de Windows Vista, Windows XP con Service Pack 2 ó superior, ó Windows Server 2003.

Si tiene Visual Studio 2005...

Primero debe instalar también el **Windows SDK**. A continuación debe instalar el **.NET Framework 3.0** (que vendrá incluido como parte de Windows Vista).
<http://msdn.microsoft.com/netframework/downloads/updates>

En principio Ud. no necesita de Visual Studio para compilar y ejecutar aplicaciones de WPF (al igual que tampoco lo necesita para desarrollar aplicaciones .NET). Cuando Ud. se descarga el SDK este viene con aplicación de línea de comando **MSBuild** con la cual Ud. podrá compilar y

generar aplicaciones WPF. Sin embargo este curso lo ilustramos sobre la base de utilizar Visual Studio porque esto nos hará a todos el trabajo infinitamente más fácil de modo que no vamos a abrumarlo en este curso explicaciones de cómo hacer las cosas con MSBuild cuando posiblemente la mayoría no lo necesitará.

Si al momento de interactuar con este curso ya dispone del nuevo Visual Studio para WinFX (código interno **Orcas**) posiblemente no tenga que instalar nada más. Por ahora suponemos que Ud. debe tener instalado **Visual Studio 2005** y además debe instalar **Visual Studio Extensions for .NET Framework 3.0 (WCF&WPF)**. Esto le añadirá a Visual Studio las herramientas que incluyen la integración de XAML con el editor Intellisense, las plantillas (templates) de proyectos para WPF, para WCF y la integración con la documentación del Windows SDK.

De momento, estas extensiones incluyen también a Visual Studio un diseñador visual, al estilo del que tenemos para las Windows Forms, pero que está aún en un estado rústico y experimental. Pero no se alarme que esta claro que un buen diseñador vendrá. De modo que por esta razón, y para que Ud. gane un dominio mayor del tema, los ejemplos de este curso los vamos a escribir directamente a mano en XAML, aunque claro que el Intellisense nos ayudará y nada nos impide echar mano también al viejo método de copiar y pegar (para lo que Ud. podrá descargar todos los proyectos de los ejemplos que vienen con el curso).

Si ya lo tiene todo instalado y dio una lectura a la lección anterior entonces manos a la obra con WPF. En esta lección vamos a ver los tres modos de crear aplicaciones en WPF, primero veremos como desarrollar una aplicación que ejecutará como aplicación de escritorio Windows, luego veremos cómo desarrollar una aplicación similar pero para que sea ejecutada por un navegador y por último veremos una forma híbrida de cómo ejecutar una aplicación Windows pero con capacidades de navegación como las que se ofrecen en las aplicaciones ejecutadas bajo un navegador.

Si tiene Visual Studio 2008...

Visual Studio 2008 y Microsoft .NET Framework 3.5, ya vienen preparados para utilizar WPF. Si dispone de este entorno de desarrollo, podrá utilizar WPF desde ya.

Aún y así, si tiene Visual Studio 2008, es recomendable que instale los últimos Service Packs que encontrará en la página Web oficial de descargas de Microsoft en este enlace. <http://download.microsoft.com/>

2.2 Crear una aplicación WPF para ejecutar como aplicación de escritorio

Abra Visual Studio y escoja la opción **New Project** para crear un nuevo proyecto. Seleccione la opción **.NET Framework 3.0** en la ventana aparecerán las plantillas para las nuevas posibilidades que da WPF tal y como se muestra en la Figura 2- 1.

Escoja la opción **Windows Application (WPF)** para indicar que queremos crear un ejecutable para una aplicación Windows con WPF (después veremos cómo podemos seleccionar también **XAML Browser Application (WPF)** si lo que queremos es crear una aplicación para usarla dentro de un navegador web). Déle un nombre al proyecto y pulse **OK**. Los ficheros de proyecto y los manifiestos de la aplicación serán creados automáticamente.

Abra el **Solution Explorer** y mire los ficheros generados. Abra el fichero Window1.xaml para que vea la plantilla en XAML creada para la interfaz (Figura 2- 2) y abra el fichero Window1.xaml.cs para que vea la plantilla C# creada para el *code-behind*

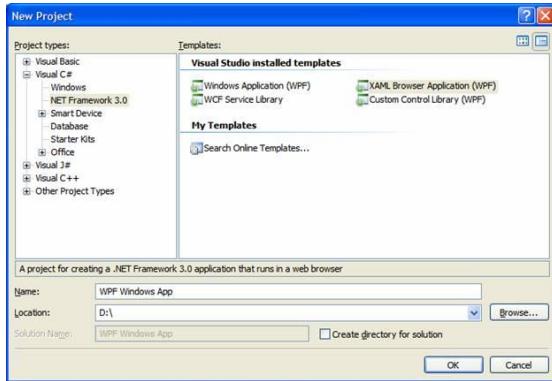


Figura 2- 1 Escogiendo la opción de crear un proyecto para aplicación de escritorio WPF

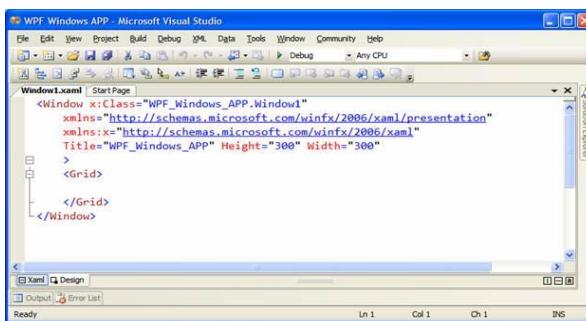


Figura 2- 2 Plantilla para el código XAML de una aplicación de escritorio WPF

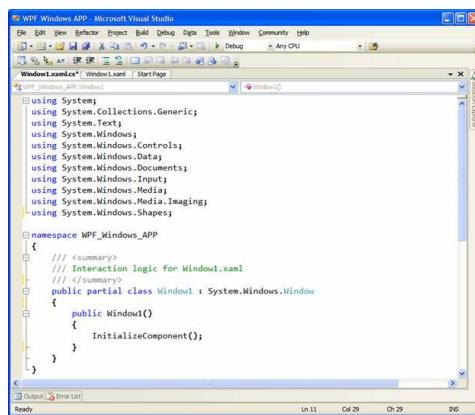


Figura 2- 3 Plantilla de código C# para completar el code-behind

Nuestro trabajo ahora consiste en llenar la plantilla con el código XAML (Figura 2- 2) y la plantilla con el código C# (Figura 2- 3), respectivamente, para lograr la interfaz deseada y la lógica de la aplicación que ésta interfaz tiene detrás. Trabajo, que si Ud. no reúne ambas cualidades, ahora podrían hacer por separado el diseñador “artista” y el programador conocedor de la lógica del negocio.

Fíjese en la plantilla de código XAML, note que el elemento Window es la raíz del árbol de contenidos y que dentro de éste se incluye un elemento panel <Grid></Grid>. Visual Studio pone implícitamente un Grid porque éste es el panel que más se utiliza debido a su versatilidad (el panel Grid se explicará en la Lección **Layout**). Por lo pronto en éste primer proyecto borre el <Grid></Grid> y coloque dentro de los tags <Window...> </Window> el código XAML del botón (Listado 2- 1).

Listado 2- 1 Código XAML de un botón

```
<Button Width="200" Height="100" FontFamily="Consolas"
       FontSize="25" Background ="Blue" Foreground="White">
    Púlsame
</Button>
```

Abra ahora el fichero Window1.xaml.cs y rellene el código C# para escribir el manejador del clic del botón con el mismo código que se utilizó en el Listado 1- 3.

Note que el código C# para declarar a la variable miBoton de tipo Button no aparece por ninguna parte. Cuando se compila una vez el código XAML en el que se hace Name="miBoton" el Intellisense incorporará esta información, de modo que a partir de ahí cuando se use la variable miBoton en el código C# este ofrecerá las facilidades acostumbradas desplegando las características asociadas al tipo de la variable (Figura 2- 4).

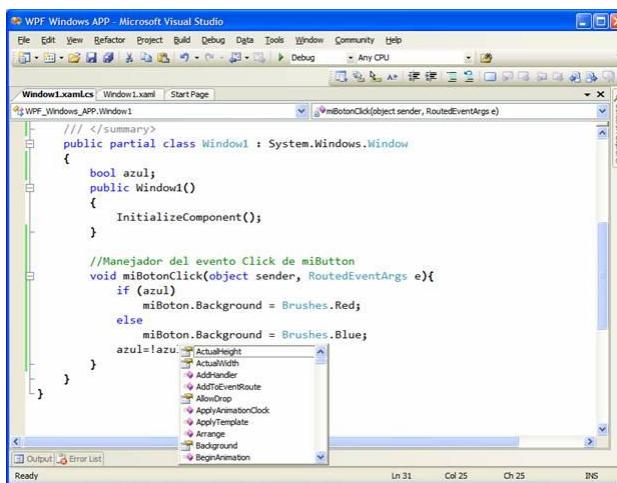


Figura 2- 4 Escribiendo el code-behind del manejador del evento clic

Busque la opción **Build** y compile el proyecto y luego ejecute la aplicación, deberá desplegarse una ventana como la que se mostró en la Figura 1- 2. Haga clic sobre el botón y observe cómo éste reacciona al evento y cambia de color.

2.3 Crear una aplicación WPF para ejecutar en un navegador

En lugar de crear una aplicación WPF que funcione como aplicación de escritorio Windows, puede crearse una aplicación WPF para que sea ejecutada a través de un navegador. Para ello siga los mismos pasos que para crear la aplicación de escritorio pero escoja la plantilla **WinFX Web Browser Application** a la hora de escoger el tipo de proyecto (Figura 2- 1).

Abra el fichero Page1.xaml, compruebe que a diferencia del caso anterior para una aplicación Windows, ahora en la plantilla aparece ahora el elemento XAML <Page> en lugar de <Window> (Figura 2- 2).

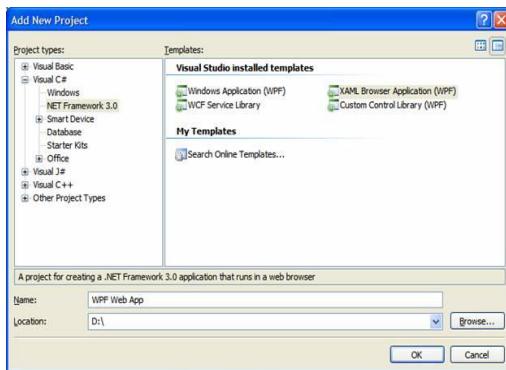


Figura 2- 1 Creando proyecto para ejecutar en un navegador

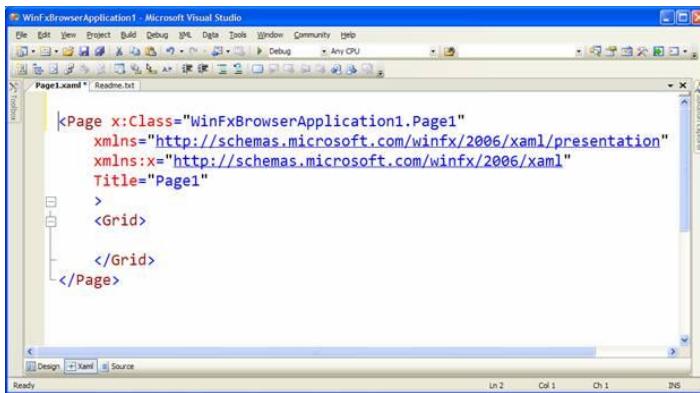


Figura 2- 2 Plantilla de XAML para aplicación WEB

Siga los mismos pasos que hicimos para la aplicación de escritorio Windows y complete el código XAML y el código C#. Fíjese que la plantilla (Figura 2- 3) para el code-behind es similar a la de la aplicación Windows pero con la diferencia de que la clase se llama Page1 y hereda de la clase System.Windows.Controls.Page.

```

WPF Windows APP - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Tools Window Community Help
Page1.cs Page1.xaml Reader.txt StartPage
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPF_Web_App
{
    /// <summary>
    /// Interaction logic for Page1.xaml
    /// </summary>
    public partial class Page1 : System.Windows.Controls.Page
    {
        public Page1()
        {
            InitializeComponent();
        }
    }
}

```

Figura 2- 3 Código C# de aplicación WPF para ejecutar en navegador

Si ejecutamos esta aplicación esta se desplegará dentro del Internet Explorer (Figura 2- 4). Note los dos nuevos botones de navegación que aparecen dentro de la ventana del navegador. Estos botones sirven para la navegación entre las páginas de la misma aplicación y son puestos automáticamente por WPF, por haber seleccionado la opción de generar una aplicación **Web Browser**. Además como siempre están los botones del Internet Explorer para navegar entre distintas páginas abiertas con Internet Explorer.

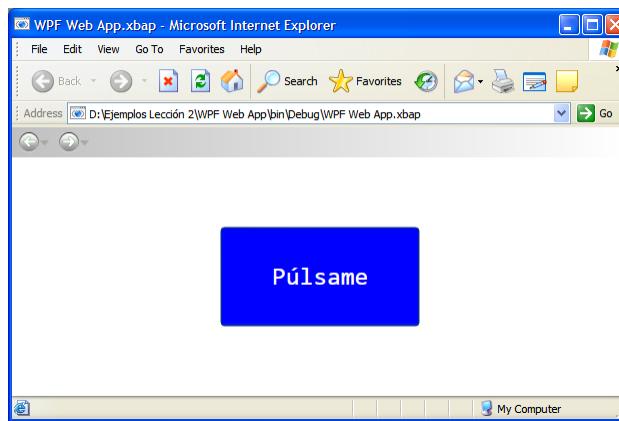


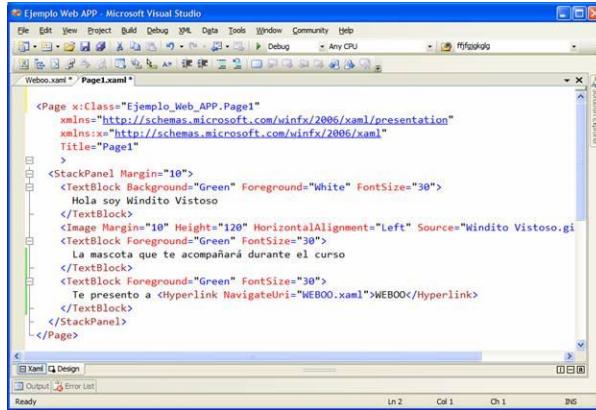
Figura 2- 4 Aplicación WPF ejecutando en el navegador

Para el ejemplo anterior los botones de navegación no son de utilidad porque tenemos una sola página y por tanto no hay páginas entre las que navegar (note que en la imagen los botones aparecen como deshabilitados).

Sin embargo si tuviéramos el proyecto con los dos ficheros Windito.xaml (Figura 2- 5) y WEBOO.xaml (Figura 2- 6) si habría navegación entre páginas y os botones nos sirven para ello.

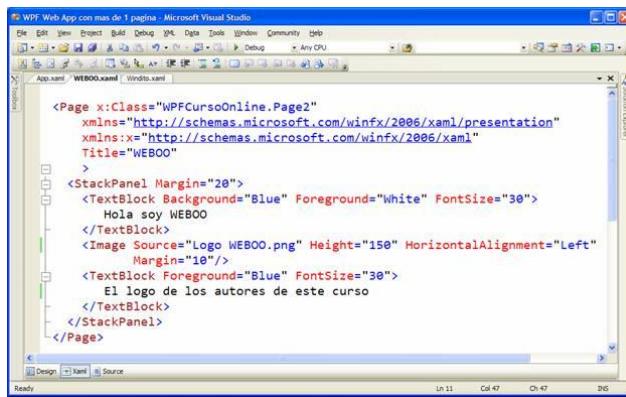
Note en el XAML de la página de la Figura 2- 5 que hemos utilizado un nuevo elemento XAML que es **Hyperlink**, este nos permite enlazar una página con otra cuya ubicación hay que darle como valor al atributo **NavigateUri**. Compile y ejecute la aplicación y se desplegará dentro del Internet Explorer la primera página (Figura 2- 7). De clic sobre el hyperlink y se pasará a la página siguiente (Figura 2- 8). Fíjese que ahora el botón de navegación aparece activado

habilitando el movimiento sobre el historial de páginas de la misma aplicación, permitiendo en este caso regresar a la página anterior.



```
<Page x:Class="Ejemplo.Web_APP.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Page1">
    >
    <StackPanel Margin="10">
        <TextBlock Background="Green" Foreground="White" FontSize="30">
            Hola soy Windito Vistoso
        </TextBlock>
        <Image Margin="10" Height="120" HorizontalAlignment="Left" Source="Windito Vistoso.gif"/>
        <TextBlock Foreground="Green" FontSize="30">
            La mascota que te acompañará durante el curso
        </TextBlock>
        <TextBlock Foreground="Green" FontSize="30">
            Te presento a WEBOO.xaml!</TextBlock>
    </StackPanel>
</Page>
```

Figura 2- 5 Código XAML de una página con hyperlink



```
<Page x:Class="WPFCursoOnline.Page2"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="WEBOO">
    >
    <StackPanel Margin="20">
        <TextBlock Background="Blue" Foreground="White" FontSize="30">
            Hola soy WEBOO
        </TextBlock>
        <Image Source="Logo WEBOO.png" Height="150" HorizontalAlignment="Left"
              Margin="10"/>
        <TextBlock Foreground="Blue" FontSize="30">
            El logo de los autores de este curso
        </TextBlock>
    </StackPanel>
</Page>
```

Figura 2- 6 Código XAML de página referida con el hyperlink

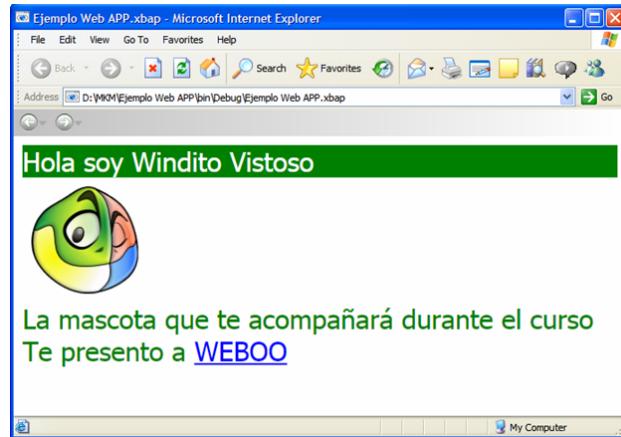


Figura 2- 7 Primera página (con hyperlink)

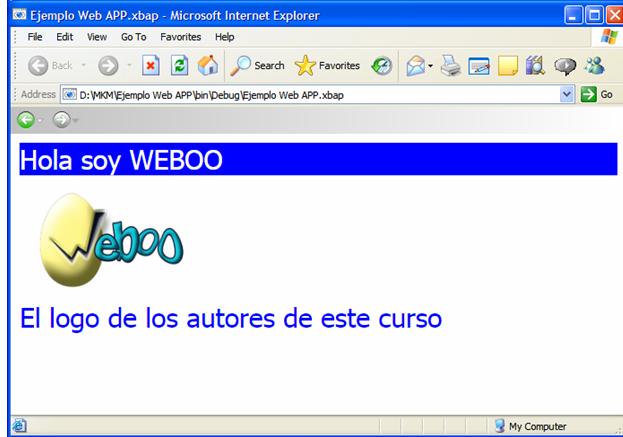


Figura 2- 8 Página con botón de navegación hacia atrás activado

2.4 Crear una aplicación WPF híbrida (Windows + Paginado)

WPF nos permite implementar una interfaz de usuario basada en páginas pero que ejecute como una aplicación de escritorio ofreciendo con ello la capacidad de poder mezclar las bondades de las aplicaciones Windows y de las aplicaciones Web.

Para crear el proyecto escoja la plantilla **WinFX Windows Application** igual que cuando va crear una aplicación Windows. Note que VS nos genera dentro del proyecto el fichero Window1.xaml que contiene a su vez la plantilla de código XAML para una ventana tal y como se muestra en la Figura 2- 13

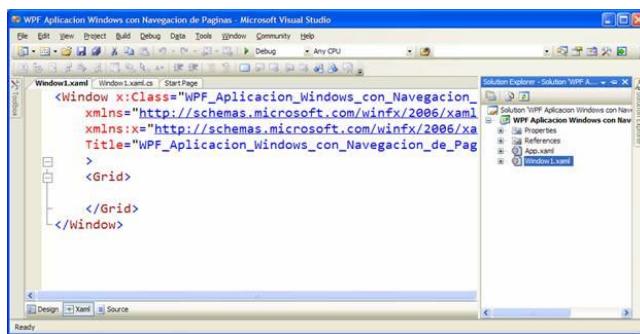


Figura 2- 13 Plantilla XAML para definir una ventana

Este fichero no nos hace falta por lo que lo podemos eliminar del proyecto. Se añaden ahora al proyecto un fichero XAML por cada página que se quiera incluir en la aplicación. En nuestro caso incorporaremos al proyecto los ficheros Windito.xaml y WEBOO.xaml utilizados en el ejemplo de la sección anterior y que consisten en un elemento Page cada uno.

Ahora hay que indicar que la aplicación que se va a ejecutar como aplicación de escritorio debe comenzar la ejecución en una página. Para ello abra el fichero App.xaml e indique en la propiedad StartupUri el nombre del fichero XAML de esa página (Figura 2- 14). Note que para ello le hemos dado el valor StartupUri="Windito.xaml".

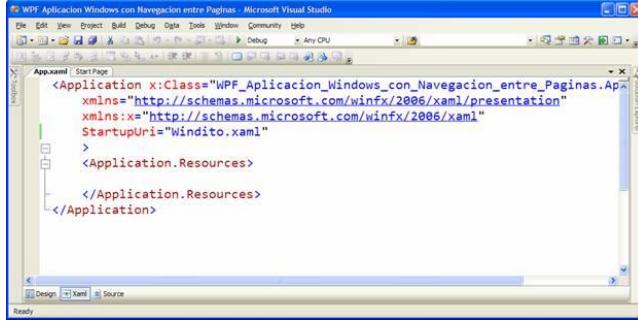


Figura 2- 14 Indicando la página inicial dentro de la aplicación Windows

Compile y ejecute la aplicación, compruebe que se mostrará una ventana en el estilo de una aplicación de escritorio Windows, es decir que no ejecutará dentro de un navegador. Sin embargo, note que ahora en su interior se ha desplegado la página y que en la parte superior de la página WPF nos ha puesto automáticamente los botones de navegación (Figura 2- 15).



Figura 2- 15 Aplicación Windows con navegación entre páginas

Aunque no estamos dentro de un navegador Web si hacemos clic en el hiperlynk WEBOO de la Figura 2- 15 navecaremos hacia la próxima página (Figura 2- 16), note como el botón de navegación está activado

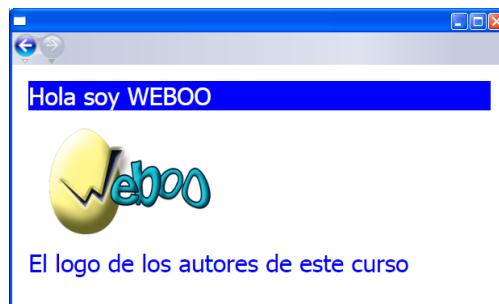


Figura 2- 16 Aplicación Windows con Navegación entre páginas

Capítulo II CONCEPTOS BÁSICOS

En este capítulo se estudian los conceptos básicos como paneles, controles, eventos, así como el enlace que alimenta y permite relacionar estos elementos. Se trata el concepto de panel y cómo se distribuyen los elementos dentro del panel. Los paneles constituyen la pieza fundamental para organizar la apariencia visual de la aplicación. También se presentan aquí los controles básicos así como se ilustran algunos de los diferentes eventos asociados a estos controles básicos y los nuevos mecanismos de propagación de eventos que introduce WPF. Finalmente con Enlace a Datos nos muestra cómo se puede hacer el enlace con datos para llenar y personalizar los controles con información.

Lección 3 Layout

La información que muestra una aplicación se agrupa y distribuye visualmente según el significado y apariencia que quiera dar la aplicación. Se conoce como **layout** a la forma en que se hace esta distribución en el área de visualización. En WPF el layout de una aplicación se maneja a través de distintos tipos de **paneles**.

Un panel es un contenedor que conoce cómo debe distribuirse visualmente su contenido en el área disponible para el panel. Su función es organizar los elementos que contiene. La forma de usar los paneles es simple pues solo es necesario colocar dentro de éste cada elemento hijo al cual se le quiere manejar su layout. El mecanismo de layout en WPF es sencillo, no obstante puede ser combinado y personalizado para conformar un sistema de layout más complejo. WPF brinda un conjunto flexible de elementos para trabajar con un layout.

Los principales paneles de WPF son:

- StackPanel**
- DockPanel**
- Grid**
- WrapPanel**
- Canvas**

Los elementos contenidos en un panel son guardados en la propiedad Children de la clase Panel que es de tipo UIElementCollection, esta propiedad es una colección a la que se asigna el conjunto de todos los elementos contenidos en el panel (estos elementos son de tipo UIElements o elementos de interfaz de usuario).

Los elementos que se pueden poner en un panel son de tipo UIElement (o elemento de interfaz de usuario). Los controles (herederos de la clase Control) son tal vez el caso más conocido de elementos de interfaz de usuario pero no son los únicos. Los controles se

estudian en la Lección **Controles**. Los paneles son a su vez elementos de interfaz lo que significa que podemos tener paneles dentro de paneles formando layouts con cualquier nivel de complejidad. Además de controles y paneles en WPF se tienen otros elementos de interfaz de usuario como figuras, decoradores, imágenes, etc. que son estudiados en diferentes lecciones de este curso.

Estudiemos a continuación cada uno de los tipos de paneles.

3.1 StackPanel

El StackPanel es un panel muy simple pero a la vez muy utilizado, en su interior los elementos se van apilando en el orden en que aparecen escritos en el código XAML (o en el orden en que hayan sido añadidos a la colección de hijos del panel en el caso en que esto se haga desde código ejecutable .NET). Los elementos se sitúan en el panel según se indique en la propiedad Orientation esto puede horizontalmente (Horizontal) o verticalmente (Vertical) que es el valor predeterminado si no se indica explícitamente la orientación indique. Por lo general un StackPanel se usa como elemento contenido en otro panel, realmente no existen muchos escenarios en los que toda la información a mostrar se quiera ver con la distribución de un StackPanel.

El código del Listado 3-1 provocará que los elementos (en este caso tres botones) se muestren de arriba abajo uno a continuación del otro según el orden en que aparecen en el XAML, ya que a la propiedad se le ha dado el valor "Vertical". Esto se muestra en la Figura 3-1.

Si en lugar "Vertical" se hubiera dado valor "Horizontal" a la propiedad Orientation entonces el contenido del panel se mostraría como en la Figura 3-2.

Listado 3- 1 Botones en un StackPanel

```
<Window x:Class="StackPanelDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Layout: StackPanel" Height="300" Width="300" >
<StackPanel Orientation="Horizontal">
    <Button Width="100" Height="50"
        Background ="AliceBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold">
        Aceptar
    </Button>
    <Button Width="100" Height="50"
        Background ="AliceBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold">
        Cancelar
    </Button>
</StackPanel>
</Window>
```

```

</Button>
<Button Width="100" Height="50"
    Background ="AliceBlue" FontFamily="Arial"
    FontSize="20" FontWeight="Bold">
    Omitir
</Button>
</StackPanel>
</Window>

```

Esperamos que entienda de manera intuitiva las propiedades que hemos utilizado dentro de cada elemento Button como Background, FontFamily, FontSize y FontWeight. De todos modos los controles se estudian en la Lección **Controles**.

El valor que se le da a propiedades como Margin o cualquier otra que represente una posición relativa a la ubicación, serán relativas a la posición en que un elemento haya sido ubicado en el Panel que lo contiene.

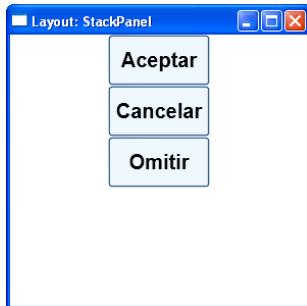


Figura 3 - 1 StackPanel con orientación vertical



Figura 3 - 2 StackPanel con orientación horizontal

3.2 DockPanel

A diferencia de un StackPanel en un DockPanel los elementos pueden situarse a la izquierda, a la derecha, arriba o abajo o de abajo en el área del panel y se puede indicar el tamaño de espacio que queremos ocupe. Los elementos van siendo ubicados en el mismo orden en que van apareciendo en el XAML.

Su funcionamiento es parecido al que brinda la propiedad Dock (acoplar) en WindowsForms.

De modo predeterminado el último control añadido al DockPanel ocupa todo el espacio que quede disponible en el panel. Esto parece ser el escenario más común. De todos modos esto puede especificarse de manera explícita dando el valor "True" o "False" a la propiedad LastChildFill .

Observe en el código XAML del Listado 3-2 que se usa un DockPanel para agrupar una imagen y dos supuestos (pues aun no le damos funcionalidad) botones de navegación, el resultado se muestra en la Figura 3-3. Note que primero ubicamos un botón a la izquierda y luego el otro a la derecha y como la imagen se ubica de último y no se le indica una ubicación esto significa que se le dará a ésta todo el espacio disponible luego de ubicar a ambos botones.

Listado 3- 2 DockPanel con Botones e Imagen como contenido

```
<DockPanel LastChildFill="True" Background="AliceBlue">
    <Button DockPanel.Dock="Left" Width="50" Height="50"
        Background ="CornflowerBlue" FontFamily="Webdings"
        FontSize="20" FontWeight="Bold"
        Foreground="White" Margin="10,0,0,0">
        9
    </Button>
    <Button DockPanel.Dock="Right" Width="50" Height="50"
        Background ="CornflowerBlue" FontFamily="Webdings"
        FontSize="20" FontWeight="Bold"
        Foreground="White" Margin="0,0,10,0">
        :
    </Button>
    <Image Source="Foto.jpg" Margin="10"></Image>
</DockPanel>
```

Note que en el código aunque hemos asignado como contenido de los botones los caracteres "9" y ":" el FontFamily seleccionado fue Webdings y por tanto se mostrarán los íconos deseados como contenido de los botones. Para más detalle vea mapa de caracteres de Windows.

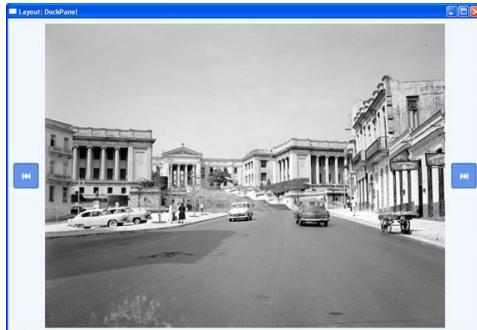


Figura 3 - 3 DockPanel (Mi álbum retro)

También podemos combinar paneles uno dentro de otro, vea en el Listado 3-3 como los botones han sido agrupados en un StackPanel al que a su vez se le ha dado la posición "Bottom" del DockPanel. Una vez mas como el StackPanel con los botones ha sido ubicado primero (en el fondo), la imagen ocupará todo el espacio restante.

Listado 3- 3 DockPanel con StackPanel e imagen como contenido

```

<DockPanel LastChildFill="True" Background="AliceBlue">
    <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
        HorizontalAlignment="Center" Margin="10"
        Background="Red">
        <Button Width="50" Height="50" Background = "CornflowerBlue"
            FontFamily="Webdings" FontSize="20"
            FontWeight="Bold" Foreground="White">
            9
        </Button>
        <Button Width="50" Height="50" Background = "CornflowerBlue"
            FontFamily="Webdings" FontSize="20"
            FontWeight="Bold" Foreground="White">
            :
        </Button>
    </StackPanel>
    <Image Source="Foto.jpg" Margin="10"></Image>
</DockPanel>

```

Para que usted note cómo se ha ubicado el StackPanel dentro del DockPanel le hemos dado valor "Red" a la propiedad Background del StackPanel. Vea el resultado en la Figura 3-4 con el área roja detrás de los botones.

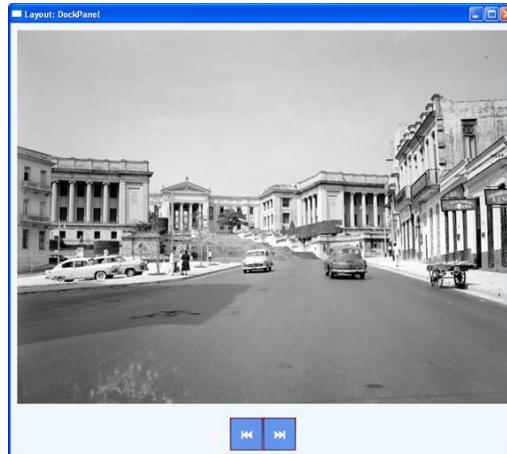


Figura 3 - 4 DockPanel con StackPanel

3.3 Grid

El Grid (Rejilla) es considerado el panel más flexible y poderoso dentro de WPF, con el que puede lograrse cualquier distribución visual de una cantidad cualquiera de elementos. De hecho

los efectos de los restantes paneles pueden lograrse con un Grid y estos solo se incluyen en WPF para mayor facilidad de algunas distribuciones específicas.

Haciendo uso del Grid podemos lograr un efecto de "enrejillado" y ubicar los elementos en celdas, agrupándolos en las columnas y filas que se definen en el Grid (un elemento puede ocupar varias celdas de este enrejillado). Para definir las columnas y las filas que forman el Grid se utilizan las propiedades `ColumnDefinition` y `RowDefinition` que son de tipo `ColumnDefinitionCollection` y `RowDefinitionCollection` y en las cuales se guarda la colección de columnas (objetos `ColumnDefinition`) y filas (objetos `RowDefinition`) que conforman el Grid. El código a continuación define un Grid de dos filas y tres columnas.

```
<Grid>
<Grid.ColumnDefinitions>
    <ColumnDefinition>
    <ColumnDefinition>
    <ColumnDefinition>
</Grid.ColumnDefinitions>

<Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
</Grid.RowDefinitions>
</Grid >
```

El ancho y altura de una columna o de una fila puede definirse con las propiedades `Width` y `Height` de `ColumnDefinition` y `RowDefinition` respectivamente. Por defecto cuando no se especifica ningún ancho ni alto se les da a todas las celdas el mismo tamaño y tanto su ancho como su altura es representado por "*".

Para colocar un elemento dentro del Grid, basta con añadirlo como un hijo de este (definirlo dentro de éste en el código XAML) e indicar en cuál columna y fila se ubica mediante las propiedades agregadas `Grid.Column` y `Grid.Row` del Grid que lo contiene.

Las dimensiones de las celdas de un Grid pueden ser modificadas, si queremos que una celda del Grid se ajuste al tamaño del elemento que contiene se debe dar el valor "Auto" al `Width` de la columna, o al `Height` de la fila. Para referirse al espacio que quede disponible dentro de una celda se usará "*" con lo que se estará indicando que este espacio sea utilizado proporcionalmente. Por tanto podemos "jugar" un poco con esta representación del espacio disponible, por ejemplo en un código como el del Listado 3-4,

Listado 3- 4 Grid, ancho de las columnas

```
<Grid ShowGridLines="True" HorizontalAlignment="Center"  
      VerticalAlignment="Center" Background="AliceBlue"  
      Height="300" Width="300">  
  <Grid.ColumnDefinitions>  
    <ColumnDefinition Width="Auto"/>  
    <ColumnDefinition Width="*"/>  
    <ColumnDefinition Width="2*"/>  
  </Grid.ColumnDefinitions>  
  <Image Height="150" Grid.Column="0" Source="Windito.gif"/>  
  <Image Height="150" Grid.Column="1" Source="Windito.gif"/>  
  <Image Height="150" Grid.Column="2" Source="Windito.gif"/>
```

Provocaría que la primera columna del Grid adopte como su ancho el ancho del elemento que conforma su contenido (en el ejemplo la imagen de ancho 150), luego el espacio restante se distribuye como ilustra la Figura 3-5.

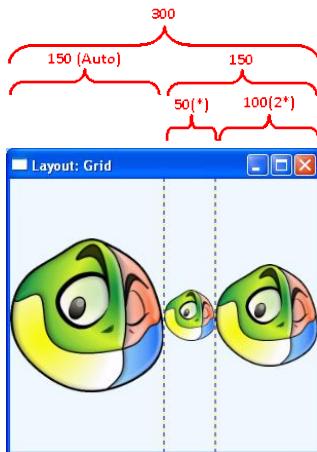


Figura 3 - 5 Grid con tres columnas

Que a la primera columna se le indique por ancho "Auto" quiere decir que esta tomará como ancho el necesario para contener al elemento que la ocupa y al indicar "*" y "2*" quiere decir que el espacio que ha quedado disponible en el Grid luego de la primera ocupar el ancho necesario, se distribuirá entre las dos restantes columnas asignando 1/3 a la primera y 2/3 a la segunda.

Si se quieren mostrar las líneas que determinan los bordes de las filas y columnas de un Grid se debe dar valor "True" a la propiedad ShowGridLines.

Veamos el código (Listado 3-5) para la interfaz visual de una calculadora como ejemplo de uso de un Grid. Se han definido cinco columnas dentro del Grid a las cuales se les ha asignado como ancho el valor "Auto" para que este se ajuste al ancho del elemento que contenga.

Listado 3- 5 Un Grid para una calculadora

```
<Grid HorizontalAlignment="Center"
      VerticalAlignment="Center"
      Background="AliceBlue">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <TextBlock Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="5"
              Background ="Black" FontFamily="Arial" FontSize="30"
              FontWeight="Bold" Foreground="White" Margin="5">
        0
    </TextBlock>
    <Button Grid.Column="0" Grid.Row="1" Height="50" Width="50"
            Background ="CornflowerBlue" FontFamily="Arial"
            FontSize="20" FontWeight="Bold" Foreground="White">
        7
    </Button>
    <Button Grid.Column="1" Grid.Row="1" Height="50" Width="50"
            Background ="CornflowerBlue" FontFamily="Arial"
            FontSize="20" FontWeight="Bold" Foreground="White">
        8
    </Button>
    <Button Grid.Column="2" Grid.Row="1" Height="50" Width="50"
            Background ="CornflowerBlue" FontFamily="Arial"
            FontSize="20" FontWeight="Bold" Foreground="White">
        9
    </Button>
```

```

</Button>
<Button Grid.Column="0" Grid.Row="2" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    4
</Button>
<Button Grid.Column="1" Grid.Row="2" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    5
</Button>
<Button Grid.Column="2" Grid.Row="2" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    6
</Button>
<Button Grid.Column="0" Grid.Row="3" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    1
</Button>
<Button Grid.Column="1" Grid.Row="3" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    2
</Button>
<Button Grid.Column="2" Grid.Row="3" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    3
</Button>
<Button Grid.Column="0" Grid.Row="4" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    0
</Button>
<Button Grid.Column="1" Grid.Row="4" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    +/-
```

```

        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">

    </Button>
    <Button Grid.Column="3" Grid.Row="1" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
    /
</Button>
<Button Grid.Column="3" Grid.Row="2" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
*
</Button>
<Button Grid.Column="3" Grid.Row="3" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
-
</Button>
<Button Grid.Column="3" Grid.Row="4" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
=
</Button>
    <Button  Grid.Column="4"  Grid.Row="3"  Grid.RowSpan="2"
Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
+
</Button>
<Button Grid.Column="4" Grid.Row="1" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="White">
C
</Button>
<Button Grid.Column="4" Grid.Row="2" Height="50" Width="50"
        Background ="CornflowerBlue" FontFamily="Symbol"
        FontSize="20" FontWeight="Bold" Foreground="White">
Ö
</Button>
</Grid>
```

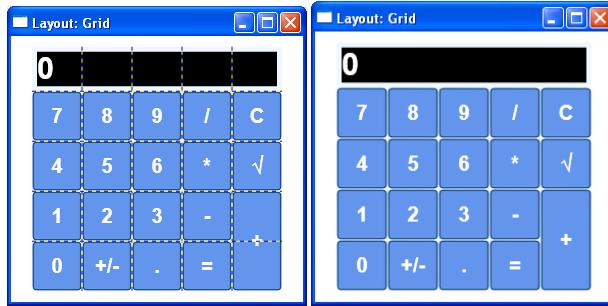


Figura 3 - 6 Calculadora en un Grid

En la Figura 3-6 a la derecha se muestra el resultado de la ejecución del código del Listado 3-5 . La Figura 3-6 a la izquierda muestra las líneas que conforman el Grid para una mejor comprensión de lo que provocan ciertas instrucciones en el código XAML.

Con la propiedad Grid.ColumnSpan, un elemento indicará sobre cuantas columnas se extenderá dentro del Grid que lo contiene. Note en la Figura 3-6 como las celdas del Grid que hacen de pantalla de la calculadora se extienden a través de las cinco columnas). Igualmente con la propiedad Grid.RowSpan se puede indicar sobre cuántas filas se extenderá (en este caso el botón "+" se extiende entre dos filas).

3.4 WrapPanel

El WrapPanel agrupa los elementos de izquierda a derecha (o de arriba abajo) pero cambiando de fila (de columna) cuando se llega el tope derecho (al fondo) del área del panel. Su funcionamiento es parecido a como se maneja el layout al escribir un documento en Word. En un WrapPanel cuando una fila de elementos ha llenado el espacio horizontal disponible (un resultado equivalente puede lograrse en sentido vertical), entonces ubica al próximo elemento al inicio de una próxima fila. El código del Listado 3-6 provoca que un conjunto de imágenes se coloquen de izquierda a derecha en el mismo orden en que aparecen en el XAML que describe al panel. Al igual que en un StackPanel el sentido en que son colocados lo define el valor "Horizontal" (el predeterminado) o "Vertical" que se le asigne a la propiedad Orientation .En la Figura 3-6 se muestra el resultado de la ejecución de este código. Un resultado semejante pero en el sentido vertical puede lograrse dando precisamente valor "Vertical" de la propiedad Orientation. La selección de uno u otro sentido para la propiedad Orientation provoca que un cambio en las dimensiones de la ventana (cómo elemento contenedor del WrapPanel) implique una redistribución (de ser necesaria para las nuevas dimensiones) de los elementos en el sentido indicado.

Listado 3- 6 WrapPanel con Imagenes como contenido y orientación horizontal
<pre style="margin: 0;"><WrapPanel Orientation="Horizontal" Background="AliceBlue"> <Image Source="Windito1.gif" Margin="10" Height="100"/></pre>

```

<Image Source="Windito2.gif" Margin="10" Height="100"/>
<Image Source="Windito3.gif" Margin="10" Height="100"/>
<Image Source="Windito4.gif" Margin="10" Height="100"/>
<Image Source="Windito5.gif" Margin="10" Height="100"/>
<Image Source="Windito6.gif" Margin="10" Height="100"/>
<Image Source="Windito7.gif" Margin="10" Height="100"/>
<Image Source="Windito8.gif" Margin="10" Height="100"/>
</WrapPanel>

```

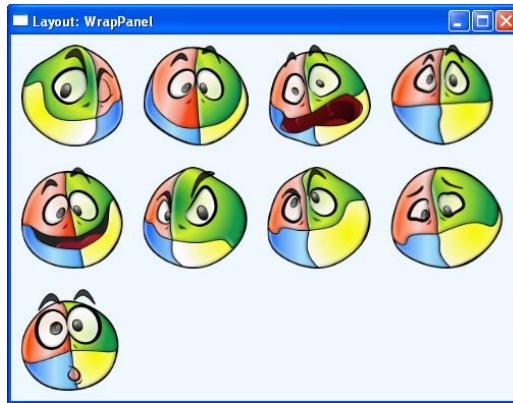


Figura 3 - 7 WrapPanel en sentido horizontal

En la Figura 3-7 solo caben las cuatro primeras imágenes (dependiendo del tamaño de la ventana) y el resto no cabe en la fila. La altura de fila es determinada por la mayor altura entre las alturas de los elementos a poner en la fila (en este ejemplo han sido imágenes de igual altura). Note que mientras sea suficiente el que quede disponible las imágenes tratan de ocupar el espacio una detrás de la otra.

Note en la Figura 3-8 a continuación el resultado de haber usado un StackPanel en lugar del WrapPanel para ubicar las imágenes. Vea como se han situado una a continuación de la otra sin ajustarse al tamaño de la ventana.

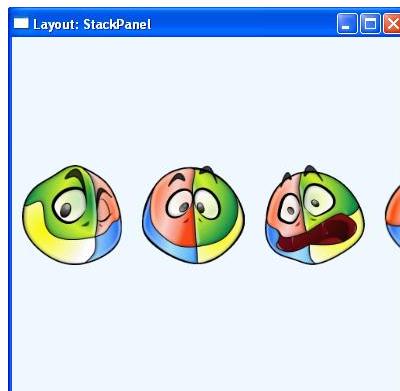


Figura 3 - 8 Usando un StackPanel el lugar del WrapPanel

Si ampliamos hacia la derecha el tamaño de la ventana podemos seguir viendo imágenes dispuestas una a continuación de la otra en el StackPanel. Imagine ahora si la cantidad de imágenes fuese suficiente como para sobrepasar el ancho de nuestra pantalla, no se estaría aprovechando todo el espacio disponible en la ventana para mostrar las imágenes .De hecho le hemos ilustrado un ejemplo donde el uso de un WrapPanel es necesario para una distribución asequible del contenido y sencilla de programar.

3.5 Canvas

El Canvas es quizás el más sencillo de todos los paneles de WPF. Cuando se quiere tener control directo sobre la posición de los elementos dentro área de visualización del panel es recomendable el uso de un Canvas. El Canvas permite posicionar elementos usando coordenadas absolutas. Para cada elemento dentro de un Canvas hay que especificar la posición que se le quiere dar, por defecto sino se indica una posición éste sería ubicado la esquina superior izquierda del panel (Top, Left).

El uso del Canvas no es recomendado cuando se desea que una aplicación sea tolerante a cambios en las dimensiones. Esto quiere decir que si un elemento se ha situado en el Canvas estableciendo sus coordenadas Top y Left (el primer TextBox del ejemplo) entonces si estrechamos o ampliamos la ventana de la aplicación ya sea por la izquierda (Left) o por la parte superior (Top) el elemento se desplazará para mantener las coordenadas relativas al par (Top, Left). Esto puede provocar que durante la reestructuración del contenido ocurra un solapamiento de los elementos contenido en el Canvas. De igual manera ocurre para los elementos cuya posición es relativa al (Right, Bottom) si el tamaño de la ventana de la aplicación es modificado desde abajo o desde la derecha.

El Listado 3- 7 muestra un ejemplo de uso de Canvas, éste simula una nevera en la que se han colocado pegatinas con recados (Figura 3-8). Los recados se han representados con controles TextBox (ver Lección **Controles**) que se han distribuido asignándole una posición específica dentro del Canvas.

Listado 3- 7 Canvas con cajas de texto e Imágenes como contenido

```
<Canvas>
    <Image Source="Nevera.jpg" Height="700"/>
    <TextBox Canvas.Top="10" Canvas.Left="10"
        Background="LightYellow" FontWeight="Bold"
        Foreground="Blue" FontSize="20"
        FontFamily="Bradley Hand ITC" AcceptsReturn="True"/>
    <TextBox Canvas.Bottom="550" Canvas.Right="10"
        Background="LightYellow" FontWeight="Bold"
        Foreground="Blue" FontSize="20"
        FontFamily="Bradley Hand ITC" AcceptsReturn="True"/>
```

```

<TextBox Canvas.Top="100" Canvas.Left="25"
    Background="LightYellow" FontWeight="Bold"
    Foreground="Blue" FontSize="20"
    FontFamily="Bradley Hand ITC" AcceptsReturn="True"/>
<TextBox Canvas.Bottom="450" Canvas.Left="250"
    Background="LightYellow" FontWeight="Bold"
    Foreground="Blue" FontSize="20"
    FontFamily="Bradley Hand ITC" AcceptsReturn="True"/>
</Canvas>

```

Cuando en el código se hace por ejemplo

```

<TextBox Canvas.Top="10" Canvas.Left="10"
    Background="LightYellow" FontWeight="Bold"
    Foreground="Blue" FontSize="20"
    FontFamily="Bradley Hand ITC" AcceptsReturn="True"/>

```

se está indicando la posición que se desea ocupe el cuadro de texto dentro del Canvas. Note cómo en XAML un elemento puede hacer referencia a una propiedad del elemento que lo contiene (padre) calificando a la propiedad con el nombre del tipo del elemento padre (Canvas.Top="10" Canvas.Left="10").

Solo es necesario especificar una posición de referencia (es decir con las propiedades Left y una Top o con las propiedades Right y Bottom). En caso de especificar (innecesariamente) ambas Top tiene prioridad sobre Bottom y Right sobre Left.

El resultado de la ejecución de este código de ejemplo se muestra en la Figura 3-9. Como puede Ud. ver hemos obtenido una nevera con la distribución de nuestras pegatinas de recados.

Los elementos aparecen ubicados dentro del Canvas según sus posiciones absolutas relativas a éste. Esto significa que los elementos que comparten un espacio común dentro del Canvas pueden solaparse. En este caso se solaparán según el orden en que aparecen contenidos dentro del panel en el código XAML que es a su vez el orden en que son renderizados.



Figura 3 - 9 Pegatinas en la nevera (Canvas)

Lección 4 Controles

Los controles son el vehículo para la interacción entre los usuarios y la aplicación. Aunque los controles tienen una manifestación visual es bueno señalar que no todo lo que conforma la interfaz visual de una aplicación tiene que ser necesariamente un control, ya que podemos tener elementos con propósitos meramente decorativos y de apariencia que no reaccionan ante ninguna acción.

La práctica en el uso de Windows ha creado un estándar de facto en cuanto a la apariencia predeterminada de los controles. Seguramente que usted está habituado a ver un botón como una suerte de rectángulo sobre el que puede hacer clic y una caja de texto como un rectángulo sobre el que se puede escribir.

Aunque la apariencia puede ser lo que primero un usuario identifica de un control, lo que realmente lo hace funcional es su comportamiento. En WPF **apariencia** y **comportamiento** son dos conceptos bien identificados e independientes.

Hasta ahora en la mayoría de las tecnologías personalizar la apariencia de un control implicaba la implementación de uno nuevo, ahora en WPF esto ya no es necesario. La capacidad de anidamiento de contenidos y el poderle asociar plantillas a los controles (ver Lección **Plantillas de Controles**) nos brinda soluciones más simples para ello.

Una de las diferencias más significativas entre WPF y Win32 es que en WPF existe una clara separación entre presentación, contenido y lógica mejor que en Win32. En WPF, probablemente Ud. no necesite escribir código ejecutable en algún lenguaje .NET (léase C# por ejemplo) para modificar la presentación de un control (al menos en la mayoría de los casos). Esto puede especificarse ahora declarativamente en el propio XAML. El contenido a presentar puede especificarse mediante enlace a datos (ver Lección **Enlace a Datos**). Incluso la lógica, o parte de

ella, puede modificarse usando diferentes elementos de su propio diseño en la plantilla y usando desencadenadores (ver Lección **Triggers**).

Una de las características más significativas de los controles en WPF destaca en un amplio grupo para los que no existen apenas restricciones en cuanto a su contenido, de acuerdo al cual pueden considerarse dos tipos de controles:

Controles que contienen un solo elemento

Estos controles son de tipo ContentControl y tienen una propiedad Content de tipo object cuyo valor es el elemento contenido en el control

Por ejemplo los tres siguientes segmentos de código XAML son equivalentes y definen a un control Button cuyo contenido es un único elemento TextBlock

```
<Button>
    Púlsame
</Button>
```

es una forma abreviada de hacer

```
<Button>
    <TextBlock>
        Púlsame
    </TextBlock>
</Button>
```

que a su vez es una forma abreviada de

```
<Button>
    <Button.Content>
        <TextBlock>
            Púlsame
        </TextBlock>
    </Button.Content>
</Button>
```

Este será convertido en un TextBlock automáticamente para formar el contenido del Button.

- **Controles que contienen una colección de elementos**

Estos controles son de tipo ItemsControl y tienen una propiedad Items de tipo ItemCollection cuyo valor es una colección de elementos contenidos dentro del control.

Si analizamos un control cualquiera de alguno de estos dos grupos (todos herederos de la clase raíz Control) notaremos que por la forma en que estos definen su contenido, éste puede ser desde una simple cadena de texto, una imagen, otro control y hasta todo un árbol de elementos

WPF. También existen controles para contenido específico (solo texto, por ejemplo TextBox) e incluso controles que no requieren de contenido alguno (por ejemplo el Thumb).

De momento la vía para interactuar desde el "exterior" con las aplicaciones WPF es a través de los controles. Los controles producen eventos debido a hacer clic en algún botón del ratón, presionar alguna tecla, o cualquier otra de las formas conocidas de entrada que seguramente usted ya conoce de Windows.

En WPF tiene existen una serie de controles básicos los cuales se estudian en esta lección. Como todo elemento de WPF, a los controles se les aplican las funcionalidades que se describen a lo largo de este curso, tales como estilos, enlace a datos, composición y soporte integrado para capacidades gráficas.

Los controles de WPF no tienen relación con los "antiguos" controles de Win32, aunque visualmente algunos se presenten predeterminadamente con la misma apariencia.

Hay propiedades comunes a los controles como Foreground, Background, Padding, etc, las cuales pueden usarse para modificar su apariencia.

A continuación un resumen de todo lo que se puede hacer en WPF con un control:

- Usar sus propiedades para modificar su apariencia y comportamiento.
- Componer controles para definir nuevos controles.
- Contener a su vez controles. Lo que se puede expresar declarativamente gracias al modelo de contenidos de XAML.
- Reemplazar la plantilla que define a un control y con ello modificar la apariencia con se que muestra el control (esto se estudia en la Lección **Plantillas de Controles**).

En lo adelante se presentan los controles más importantes y se irá mostrando parte del código XAML necesario para usarlos con figuras que muestran lo que se desplegaría en ejecución al usar el mismo. Por supuesto que es imposible presentar todos los controles ni probar toda la infinidad de propiedades de modo que esto quedará a su interés, necesidad y disponibilidad de tiempo.

4.1 Botones

Los botones son con certeza los controles más conocidos. Hay tres tipos de botones básicos: Button, RadioButton y CheckBox todos herederos (directa o indirectamente) de la clase ButtonBase.

El Listado 4-1 nos muestra un botón (un control Button) que tiene como contenido un panel StackPanel que a su vez tiene como contenido un texto y una imagen (el resultado se muestra

en la Figura 4-1). En este ejemplo se han modificado algunas de las propiedades del botón (Width, Height, Background) para variar su apariencia. A la imagen que está dentro del botón basta darle valor a una de sus propiedades Width o Height, ya que la otra se ajusta en consecuencia.

Listado 4- 1 Código XAML de ventana con un Control Button

```
<Window x:Class="Botones.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Controles Básicos: Botones" Height="300" Width="300" >
    <Button Width="200" Height="150" Background ="AliceBlue">
        <StackPanel>
            <TextBlock HorizontalAlignment="Center"
                FontFamily="Arial" FontSize="20"
                FontWeight="Bold" Foreground="Navy">
                ¿Yo, en un Botón?
            </TextBlock>
            <Image Source="WinditoAsombrado.gif" Height="100" Margin="5"/>
        </StackPanel>
    </Button>
</Window>
```



Figura 4 - 1 Ventana con botón

El Listado 4-2 nos muestra un control RadioButton que tiene un texto (TextBlock) y una imagen (Image). La Figura 4-2 muestra el resultado de la ejecución de este código.

Listado 4- 2 Código XAML de ventana con un Control RadioButton

```
<RadioButton HorizontalAlignment="Center" >
    <StackPanel Orientation="Horizontal">
        <Image Source="WinditoRadioButton.gif" Height="100"
Margin="5"/>
        <TextBlock VerticalAlignment="Center"
            FontFamily="Arial" FontSize="20"
            FontWeight="Bold" Foreground="Navy">
            Selecciónname si quieres
        </TextBlock>
    </StackPanel>
</RadioButton>
```



Figura 4 - 2 Ventana con RadioButton

Los RadioButton pueden agruparse en paneles. En el grupo de RadioButton contenidos en un panel no podrá haber más de uno seleccionado.

El botón CheckBox se usa de modo similar al RadioButton (Listado 4-3 y Figura 4-3).

Listado 4- 3 Código XAML de ventana con un Control CheckBox

```
<CheckBox HorizontalAlignment="Center" >
    <StackPanel Orientation="Horizontal">
        <TextBlock VerticalAlignment="Center"
            FontFamily="Arial" FontSize="20"
            FontWeight="Bold" Foreground="Navy">
            Selecciónname a mí
        </TextBlock>
        <Image Source="WinditoCheckBox.gif" Height="100"
Margin="5"/>
    </StackPanel>
</CheckBox>
```



Figura 4 - 3 Ventana con CheckBox

El Listado 4-4 a continuación muestra varios de los botones vistos en esta lección y hace uso de lo aprendido en la Lección **Layout** para su distribución en la ventana.

La Figura 4-4 muestra el resultado de la ejecución del código XAML del Listado 4-4.

Listado 4- 4 Código XAML de ventana con varios controles botones

```
<StackPanel>

<Grid>    <Grid.ColumnDefinitions>

    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>

    <RadioButton      Grid.Column="0"      Grid.Row="0"
Margin="10,0,0,0">
        <StackPanel Orientation="Horizontal">
            <Image Source="WinditoMolesto.gif" Height="100" Margin="5"/>
            <TextBlock VerticalAlignment="Center"
                FontFamily="Arial" FontSize="20"
                FontWeight="Bold" Foreground="Red">
                Seleccióname a mí
            </TextBlock>
        </StackPanel>
    </RadioButton>
    <RadioButton      Grid.Column="1"      Grid.Row="0"
Margin="10,0,0,0">
        <StackPanel Orientation="Horizontal">
            <Image Source="WinditoTriste.gif" Height="100" Margin="5"/>
            <TextBlock VerticalAlignment="Center"
                FontFamily="Arial" FontSize="20"
                FontWeight="Bold" Foreground="Olive">
                Seleccióname a mí
            </TextBlock>
        </StackPanel>
    </RadioButton>
    <CheckBox      Grid.Column="0"      Grid.Row="1"
Margin="10,0,0,0">
```

```

<StackPanel Orientation="Horizontal">
    <Image Source="WinditoFeliz.gif" Height="100" Margin="5"/>
    <TextBlock VerticalAlignment="Center"
        FontFamily="Arial" FontSize="20"
        FontWeight="Bold" Foreground="CornflowerBlue">
        Seleccióname a mí
    </TextBlock>
</StackPanel>
</CheckBox>
<CheckBox Grid.Column="1" Grid.Row="1"
Margin="10,0,0,0">
    <StackPanel Orientation="Horizontal">
        <Image Source="WinditoAsustado.gif" Height="100"
Margin="5"/>
        <TextBlock VerticalAlignment="Center"
            FontFamily="Arial" FontSize="20"
            FontWeight="Bold" Foreground="Gold">
            Seleccióname a mí
        </TextBlock>
    </StackPanel>
</CheckBox>
</Grid>
<StackPanel Orientation="Horizontal"
HorizontalAlignment="Center">
    <Button FontFamily="Arial" FontSize="20"
FontWeight="Bold">
        Aceptar
    </Button>
    <Button FontFamily="Arial" FontSize="20"
FontWeight="Bold">
        Cancelar
    </Button>
</StackPanel>
</StackPanel>

```

Note en la Figura 4-4 como ambos controles RadioButton no pueden seleccionarse a la vez.

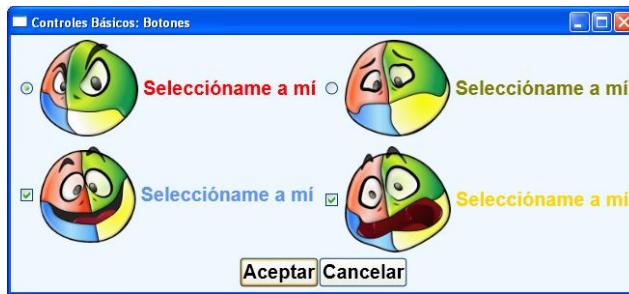


Figura 4 - 4 Ventana con varios controles botones

4.2 Slider y Scroll

El control Slider y el control Scroll heredan de la clase RangeBase ya que ambos trabajan sobre un rango de valores.

El control Slider permite seleccionar valores dentro de un rango. Por ejemplo, el control de volumen que Ud. ve en Windows Media Player o el ajuste de transparencia de color que puede encontrar en Power Point se podrían lograr usando un Slider. En el ejemplo del Listado 4-5 se hace uso de un control Slider para determinar las dimensiones de una imagen para ello fue necesario hacer uso de Enlace a Datos (Binding) (esto se estudia en más detalle en la Lección **Enlace a Datos**). La ejecución de este código se muestra en la Figura 4-5.

Listado 4-5 Código XAML de ventana con un Control Slider

```
<Window x:Class="Slider_Scroll.Window1"
    xmlns="http://schemas.microsoft.com/winf/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winf/2006/xaml"
    Title="Controles Básicos: Slider" Height="300" Width="300">
<StackPanel>
    <TextBlock Margin="10,10,0,0" HorizontalAlignment="Left">
        Ajustar tamaño de Imagen
    </TextBlock>
    <Slider Name="imageSizeSlider" Margin="10, 0, 0, 0"
        Width="100" Orientation="Horizontal" HorizontalAlignment="Left"
        Value="50" Minimum="0" Maximum="200"
        TickPlacement="BottomRight" TickFrequency="10"/>
    <Image Source="WinditoSlider.gif"
        HorizontalAlignment="Left"
        Margin="50,20,0,0"
        Height="{Binding ElementName=imageSizeSlider,Path=Value}"/>
</StackPanel>
</Window>
```

Note que a la propiedad Height de la imagen se le da el valor de la siguiente forma Height="{Binding ElementName=imageSizeSlider, Path=Value}". Este Binding asocia la propiedad Height de la imagen con el valor que tenga el control llamado imageSizeSlider que es en este caso de tipo Slider.



Figura 4 - 5 Ventana con Slider

Al igual que el Slider el control ScrollViewer trabaja asociado a un rango (Listado 4-6).

Listado 4- 6 Código XAML de ventana con un Control ScrollViewer

```
<ScrollView  
    HorizontalScrollBarVisibility="Auto"  
    VerticalScrollBarVisibility="Auto">  
    <Image Source="WinditoScroll.gif"/>  
</ScrollView>
```

Las propiedades HorizontalScrollBarVisibility y VerticalScrollBarVisibility a las que se les ha dado valor "Auto" indican que los ScrollViewer tanto horizontal como vertical solo sean visibles cuando sea necesario (es decir cuando el contenido sobrepasa las dimensiones en que se está mostrando como se ilustra en la Figura 4-6). Los otros valores posibles de estas propiedades son Disabled, Hidden y Visible.



Figura 4 - 6 Ventana con Scroll

4.3 Controles de Texto

WPF tiene una variedad de controles para mostrar y editar textos, el más sencillo de ellos es el TextBox (Listado 4-7) que probablemente ya Ud ha visto en otras lecciones. El TextBox brinda las capacidades básicas para edición: escribir un texto, dar cambios de línea (a través de la

propiedad AcceptsReturn), cortar, copiar, pegar etc. Note que lo que muestra la Figura 4-7 como contenido de los controles de texto, es el texto que el usuario de la aplicación haya escrito sobre el control ya que los controles de texto están preparados para recibir este tipo de entrada (del mismo modo que si se hubiese escrito en el XAML de la siguiente forma:

```
<TextBox.Text>  
    Texto que se quiera mostrar  
</TextBox.Text>
```

Listado 4- 7 Código XAML de ventana con un Control TextBox

```
<Window x:Class="ControlesDeTexto.Window1"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title=" Controles Básicos: Texto" Height="300" Width="300" >  
<StackPanel>  
    <TextBox Margin="10" AcceptsReturn="True" FontSize="20"/>  
</StackPanel>  
</Window>
```

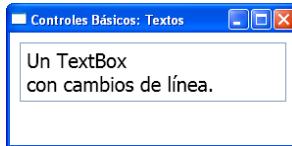


Figura 4 - 7 Ventana con TextBox

Otro control para el trabajo con textos es el PasswordBox (Listado 4-8, Figura 4-8) que a diferencia del TextBox, muestra camuflado el contenido que se teclee sobre el control. Este control en lugar de una propiedad Text para acceder al contenido, tiene una propiedad Password.

Listado 4- 8 Código XAML de ventana con un Control PasswordBox

```
<StackPanel>  
    <PasswordBox Margin="10" FontSize="20"/>  
</StackPanel>
```

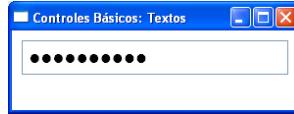


Figura 4 - 8 Ventana con PasswordBox (para los curiosos el password es: "aprendaWPF")

Por último veremos un control para la edición de texto, el control RichTextBox (Listado 4-9). Este control permite que su contenido sea algo más que simple texto, en el ejemplo vamos a escribir texto y pegar imágenes como parte del mismo contenido del RichTextBox de modo de obtener un resultado como el que nos muestra la Figura 4-9.

Listado 4-9 Código XAML Control RichTextBox

```
<StackPanel>
    <RichTextBox Margin="10" FontSize="20"/>
</StackPanel>
```



Figura 4 - 9 Ventana con RichTextBox

4.4 Etiquetas

El control Label (Listado 4-10) es un control muy utilizado para, como su nombre indica, "etiquetar" a aquellos controles que no disponen de una propiedad cuyo valor les sirva de etiqueta. Una etiqueta no tiene porque ser un simple texto, sino puede ser también cualquier elemento de interfaz, (basado en que el control Label es un ContentControl), como por ejemplo una imagen. Con las etiquetas se puede expresar la apariencia de atajos para seleccionar determinado comando, haciendo que el contenido de la etiqueta sea un texto, y a éste se le puede hacer resaltar un carácter escribiéndolo precedido de underscore (_). Note el segmento que hemos destacado en negrita dentro del XAML del Listado 4-10 delante de la letra que queremos identificar para formar parte del acceso directo

(<AccessText> _Nombre <AccessText/>).

En el código XAML del Listado 4-10 hemos vuelto a hacer uso de Binding, en este caso para asociar un TextBox a la propiedad Target del control Label. Al asociar a esta propiedad un elemento determinado (en este caso un TextBox) estamos especificando que el atajo que se define en el Label es para acceder a este elemento (Target). De este modo podemos lograr que la mayor parte del código quede expresado de forma declarativa en el XAML, con el uso de Binding se evita hacer uso del code-behind.

Listado 4- 10 Código XAML de ventana con un Control Label

```
<StackPanel>
  <DockPanel>
    <Label Name="Etiqueta1"
      Target="{Binding ElementName=Texto1}"
      VerticalAlignment="Center">
      <AccessText>_Nombre:</AccessText>
    </Label>
    <TextBox Name="Texto1" Margin="10"
      FontSize="20" VerticalAlignment="Center"/>
  </DockPanel>
  <DockPanel>
    <Label Name="Etiqueta2"
      Target="{Binding ElementName=Texto2}"
      VerticalAlignment="Center">
      <AccessText>_Apellidos:</AccessText>
    </Label>
    <TextBox Name="Texto2" Margin="10"
      FontSize="20" VerticalAlignment="Center"/>
  </DockPanel>
</StackPanel>
```

El resultado de la ejecución de este código XAML será el que se muestra en la Figura 4-10 a continuación, note como las letras (N y A) de las etiquetas se encuentran subrayadas. Si aplicamos la combinación de teclas Alt+N el cursor aparece en la primera caja de texto como muestra la Figura 4-10 a la derecha, de igual modo con Alt+A el cursor aparecería en la segunda caja de texto. Note que la correspondencia entre etiqueta y caja de texto es la especificada a través de la propiedad Target.

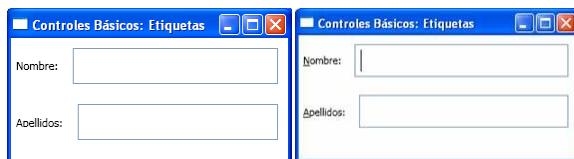


Figura 4 - 10 Ventana con Etiquetas

4.5 Selectores

Estos controles brindan una forma de selección dentro de un conjunto de elementos. Entre esta familia de controles tenemos a ComboBox, ListBox y TabControl, así como también los RadioButton cuando como vimos anteriormente se encuentran agrupados en un panel.

4.5.1 ComboBox y ListBox

Veamos el ComboBox que se define en el Listado 4-11. El resultado de la ejecución lo ilustra la Figura 4-11.

Listado 4- 11 Código XAML de ventana con un Control ComboBox <pre> <Window x:Class="Selectores.Window1" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="Controles básicos: Seleccionadores" Height="300" Width="300" > <StackPanel> <ComboBox Margin="10" SelectedIndex="0"> <TextBlock FontFamily="Arial" FontWeight="Bold"> Lección 3 Layout </TextBlock> <TextBlock FontFamily="Arial" FontWeight="Bold"> Lección 4 Controles </TextBlock> <TextBlock FontFamily="Arial" FontWeight="Bold"> Lección 5 Eventos y Comandos </TextBlock> <TextBlock FontFamily="Arial" FontWeight="Bold"> Lección 6 Enlace a Datos </TextBlock> </ComboBox> </StackPanel> </Window></pre>
--



Figura 4 - 11 Ventana con ComboBox

Un efecto semejante puede lograrse con un ListBox como muestra el Listado 4-12.

Listado 4- 12 Código XAML de ventana con un Control ListBox

```
<StackPanel>
    <ListBox Margin="10" SelectedIndex="0">
        <TextBlock FontFamily="Arial" FontWeight="Bold">
            Lección 3 Layout
        </TextBlock>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
            Lección 4 Controles
        </TextBlock>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
            Lección 5 Eventos y Comandos
        </TextBlock>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
            Lección 6 Enlace a Datos
        </TextBlock>
    </ListBox>
</StackPanel>
```

La Figura 4-12 muestra el resultado de la ejecución del código XAML del Listado 4-12.



Figura 4 - 12 Ventana con ListBox

4.5.2 TabControl

TabControl es otro control selector. Este agrupa su contenido y permite identificarlo visualmente mediante "pestañas" asociadas a su contenido. Tanto el contenido como su encabezado (lo que se muestra en la pestaña) pueden ser cualquier elemento WPF (imagen, texto, botón, etc.). Note en el Listado 4-13 como TabItem permite definir de forma sencilla su propio encabezado y el contenido.

Listado 4- 13 Código XAML de ventana con un Control TabControl

```
<StackPanel>
  <TabControl Margin="10">
    <TabItem Selector.IsSelected="True">
      <TabItem.Header>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
          Lección 3 Layout
        </TextBlock>
      </TabItem.Header>
      <Image Height="200" Source="WinditoLayout.gif"></Image>
    </TabItem>
    <TabItem>
      <TabItem.Header>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
          Lección 4 Controles
        </TextBlock>
      </TabItem.Header>
      <Image Height="200" Source="WinditoControles.gif"></Image>
    </TabItem>
    <TabItem>
      <TabItem.Header>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
          Lección 5 Eventos y Comandos
        </TextBlock>
      </TabItem.Header>
      <Image Height="200"
Source="WinditoEventosComandos.gif"></Image>
    </TabItem>
    <TabItem>
      <TabItem.Header>
        <TextBlock FontFamily="Arial" FontWeight="Bold">
```

```

Lección 6 Enlace a Datos
</TextBlock>
</TabItem.Header>
<Image Height="200"
Source="WinditoEnlaceDatos.gif"></Image>
</TabItem>
</TabControl>
</StackPanel>

```

Note en la Figura 4-13 como están relacionados "encabezado" y "contenido". Si se hace clic sobre un encabezado el control muestra el contenido que en este caso es una imagen.



Figura 4 - 13 Ventana con TabControl

4.6 Menús

Para ilustrar el uso de los controles Menu vamos a apoyarnos en dos ejemplos: uno haciendo uso del menú principal de las aplicaciones y otro de un menú de contexto. Si no logra identificar cuáles son estos, no se preocupe los reconocerá en cuanto vea los ejemplos.

Veamos primero los Menú como muestra el Listado 4-14.

Listado 4- 14 Código XAML de ventana con un Control Menu
<pre> <Window x:Class="Menus.Window1" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="Controles Básicos: Menús" Height="300" Width="300" > <StackPanel> <Menu> <MenuItem> <MenuItem.Header> <AccessText>_Edición</AccessText> </MenuItem.Header> <MenuItem Header="Cortar"/> <MenuItem Header="Copiar"/> <MenuItem Header="Pegar"/> </pre>

```

</MenuItem>
</Menu>
</StackPanel>
</Window>

```

La Figura 4-14 muestra el resultado de la ejecución del código XAML del Listado 4-14 desplegando un menú principal con las opciones de Cortar, Copiar y Pegar.



Figura 4 - 14 Ventana con Menú Principal

A cada MenuItem podemos asociarle un comando a ejecutar (como se verá en la Lección **Eventos y Comandos**). También se pueden asignar teclas de acceso a cualquiera de estos MenuItem.

También podemos crear un menú de contexto como muestra el Listado 4-15 (el que se despliega normalmente al clic derecho del ratón en las aplicaciones Windows).

Listado 4- 15 Código XAML de ventana con un Control ContextMenu

```

<Window.ContextMenu>
  <ContextMenu>
    <MenuItem Header="Cortar"/>
    <MenuItem Header="Copiar"/>
    <MenuItem Header="Pegar"/>
  </ContextMenu>
</Window.ContextMenu>

```

Se ha colocado un ContextMenu a nivel de ventana de la aplicación (Window.ContextMenu), esto implica que en cualquier lugar de la ventana en donde quiera que se de un clic derecho del ratón se desplegará el menú de contexto, como muestra la Figura 4-15. Un efecto equivalente puede lograrse a nivel de otros elementos WPF, es decir podemos definir el contexto en el cual puede desplegarse el menú (por ejemplo Grid.ContextMenu).



Figura 4 - 15 Ventana con Menú de Contexto

4.6 ToolBars

La mayoría de las aplicaciones además de un menú principal brindan una barra de herramientas. En WPF es posible definir de forma sencilla nuestra propia barra de herramientas como se muestra en el Listado 4-16. Podemos también tener varias barras de herramientas agrupadas en un **ToolBarTray**. Este control (**ToolBar**) tiene integrada la misma apariencia y funcionalidad de las barras de herramientas con las que trabajamos en Office.

Listado 4- 16 Código XAML de ventana con un Control **ToolBar**

```
<Window x:Class="ToolBar.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Controles básicos: Barra de Herramientas"
Height="300" Width="300" >
<StackPanel>
<ToolBar>
<Button><Image Source="new.bmp"/></Button>
<Button><Image Source="open.bmp"/></Button>
<Button><Image Source="save.bmp"/></Button>
<Separator/>
<Button><Image Source="cut.bmp"/></Button>
<Button><Image Source="copy.bmp"/></Button>
<Button><Image Source="paste.bmp"/></Button>
<Separator/>
<Button><Image Source="print.bmp"/></Button>
<Button><Image Source="preview.bmp"/></Button>
</ToolBar>
</StackPanel>
</Window>
```

La Figura 4-16 muestra una barra de tareas con los botones con imágenes como contenido justo como se ha especificado en el código XAML del Listado 4-16.



Figura 4 - 16 Ventana con Barra de Herramientas

4.7 Expander

El control Expander permite mostrar un encabezado que identifica un contenido que puede **contraerse y expandirse** a través del control. El Listado 4-17 nos muestra un control Expander cuyo contenido es un grupo de ComboBox. La dirección en que se expande o contrae el contenido puede especificarse a través de la propiedad ExpandDirection y sus valores pueden ser "Down", "Up", "Left" y "Right".

Listado 4- 17 Código XAML de ventana con un Control Expander

```
<Window x:Class="Expander.Window1"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Controles básicos: Expander" Height="300" Width="300" >
<StackPanel>
<Expander Background="AliceBlue"
    FontSize="20" Foreground="DarkBlue"
    HorizontalAlignment="Left"
    Header="Lección Controles Básicos"
    ExpandDirection="Down" IsExpanded="True"
    Margin="10">
<StackPanel>
    <RadioButton      IsChecked="True"      FontSize="16"
Margin="10,0,10,0">
        Excelente
    </RadioButton>
    <RadioButton FontSize="16" Margin="10,0,10,0">
        Muy Bien
    </RadioButton>
</StackPanel>
</Expander>
</StackPanel>
</Window>
```

```

</RadioButton>
<RadioButton FontSize="16" Margin="10,0,10,0">
    Bien
</RadioButton>
<RadioButton FontSize="16" Margin="10,0,10,0">
    Regular
</RadioButton>
    <RadioButton IsEnabled="False" FontSize="16"
Margin="10,0,10,0">
        No he aprendido absolutamente nada
    </RadioButton>
</StackPanel>
</Expander>
</StackPanel>
</Window>

```

Hemos deshabilitado una de las opciones para que usted no desista de aprender WPF con este curso.

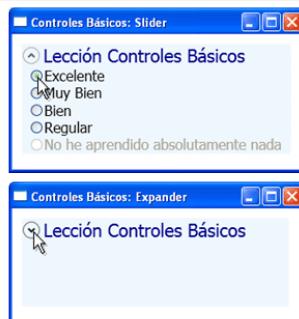


Figura 4 - 17 Ventana con Expander

Es imposible abordar en esta lección la basta variedad de posibilidades que dan los controles en WPF. Se han mostrado aquí algunas de las características más significativas de aquellos controles que podemos considerar más básicos y que son suficientes para proveer de una rica apariencia y funcionalidad a la mayoría de las aplicaciones. Lo invitamos a experimentar con ellos e ir descubriendo más capacidades.

Lección 5 Eventos y Comandos

La plataforma .NET tiene todo un mecanismo estándar definido para el trabajo con eventos. WPF hace uso de este estándar y lo enriquece con un nuevo mecanismo de disparo de eventos (utilizado sobre todo para los eventos de teclado y ratón). Los eventos en WPF siguen estrategias de propagación asociadas a que los elementos en WPF pueden estar contenidos en otros elementos.

5.1 Enrutamiento de los eventos

Veamos como ejemplo una ventana (Listado 5-1) en la que tenemos un botón cuyo contenido es a su vez una rejilla (Grid) con dos celdas, una contiene un cuadro texto y la otra una imagen. Aunque las dos celdas de la rejilla que conforman el contenido del botón sean independientes, el elemento botón como un todo debe responder a la acción de clic independientemente de si el clic ha ocurrido sobre la celda que contiene el texto o sobre la que contiene la imagen. La Figura 5-1 muestra dos de los escenarios posibles con el cursor sobre el texto y el cursor sobre la imagen.

Listado 5-1 Código XAML de Ventana con Botón con Texto e Imagen como contenido.

```
<Window x:Class="BotónTextolImagen.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Botón caja de texto e imagen" Height="300" Width="300" >
    <Button Grid.Column="0" Height="100" Width="200">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="*"/>
            </Grid.RowDefinitions>
            <TextBlock Grid.Row="0" FontSize="30">
                Desarrolla con
            </TextBlock>
            <Image Grid.Row="1" Source="MSDN.gif"/>
        </Grid>
    </Button>
</Window>
```



Figura 5 - 1 Ventana con botón (cursor sobre el texto y cursor sobre la imagen)

En el modelo estándar de .NET el manejo de los eventos consiste en que cada elemento contenido en el control responda al evento al que queremos reaccione el control. Esto implicaría en este ejemplo que tanto el bloque de texto como la imagen tendría que registrarse como manejador de dicho evento.

Imagine lo tedioso que esto podría resultar ahora en WPF registrar los manejadores de los eventos si tenemos en cuenta que prácticamente no hay restricciones sobre lo que puede conformar el contenido de un control (todo un árbol de elementos anidados dentro de otros). Afortunadamente esto no es necesario. WPF hace uso de los llamados **Eventos Enrutados** (RoutedEvent). Un Evento Enrutado en lugar de tener en cuenta solo a aquellos manejadores que se han registrado en el elemento sobre el que directamente se produce el evento, podrá llamar a todos los manejadores especificados en los elementos que están en la "ruta" del árbol en el que está incluido el elemento.

Los Eventos enrutados pueden ser hacia afuera (Bubbling), hacia adentro (Tunneling) o directos (Direct).

- ② **Eventos Bubbling:** Primero busca por manejadores en el elemento donde se ha originado el evento, luego hace lo mismo en el elemento que lo contiene (padre) y así sucesivamente hasta la raíz del árbol de elementos (que como ya sabemos del Capítulo I **Introducción** es una ventana o una página). El evento se dispara en un elemento y se propaga hacia su contenedor más inmediato que a su vez lo dispara a su contenedor y así hasta llegar al elemento raíz.

Un evento como clic de ratón se considera que se origina en un elemento cuya área de layout es la menor que contiene al punto donde estaba el cursor al hacer clic. Así en la Figura 5-1 izquierda el evento se origina sobre el TextBlock y en la figura de la derecha el evento se origina sobre el Image

- ② **Eventos Tunneling:** El proceso ocurre al inverso que con los Bubbling, busca primero manejadores en el elemento raíz para luego descender buscando en cada hijo (elemento contenido) hasta llegar al elemento en que se originó.

Tenga presente que por la forma arbórea en que están contenidos los elementos solo hay un camino para llegar de la raíz al elemento en el que se originó el evento.

- ② **Eventos directos:** El mecanismo es semejante al que se ha tratado hasta ahora en .NET (por ejemplo en las Windows Forms). Solo se tendrá en cuenta el manejador en el elemento en que se ha originado el evento. Por ejemplo un evento clic cuando el cursor está sobre el área que ocupa una imagen (Figura 5-1 a la derecha) solo se tendrá en cuenta en el manejador del elemento Image. En este caso el evento no se propaga en ningún sentido.

Para un mismo suceso, por ejemplo un clic a un botón, el orden de llamadas comienza por los eventos Tunneling, luego el directo y finalmente los eventos de Bubbling. En WPF la mayoría de los eventos con ruta (exceptuando los directos) se disparan en pares: uno Tunneling y otro Bubbling. Por convenio en WPF el nombre de los Tunneling que conforman el par comienza siempre con "Preview" seguido por el nombre del correspondiente Bubbling (por ejemplo PreviewMouseLeftButtonDown y MouseLeftButtonDown). Los eventos Tunneling son disparados primero, seguidos directamente del Bubbling correspondiente. Se pueden registrar manejadores para cada uno de los eventos tanto Tunneling como Bubbling.

Veamos un ejemplo de aplicación donde queremos conocer las coordenadas del cursor del ratón así como el tipo del elemento de la interfaz sobre el que se encuentra situado.

El Listado 5-2 muestra el código de una ventana con varios elementos hijos y cómo en XAML se asocia un manejador a un evento. En este caso se ha asociado al evento MouseMove. Este manejador asociado al MouseMove en el elemento Window se ejecutará independientemente del en el que se originó el evento MouseMove pues, será "burbujeado" hacia arriba siguiendo la ruta a la raíz del árbol que forman los elementos de la interfaz.

Gracias al mecanismo de los Eventos con Ruta, para lograr esta funcionalidad solo tenemos que registrar un manejador para este evento en el elemento padre de la jerarquía (en este caso Window). Por ser MouseMove un evento Bubbling el mecanismo de eventos de WPF se encargará de que el evento "suba" siguiendo una ruta de dónde se originó hacia su parent (su elemento contenedor) hasta llegar a la raíz (Window o Page) donde será manejado. Note que no es necesario registrarse al evento en ningún elemento intermedio.

Listado 5-2 Manejando el evento MouseMove

```
<Window x:Class="Eventos.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Eventos con Ruta" Height="300" Width="300"
    Background="AliceBlue" MouseMove="WindowMouseMove" >
<StackPanel>
    <TextBlock Margin="10" HorizontalAlignment="Right" Name="TB"
        FontSize="16" FontWeight="Bold" Foreground="Red">
        </TextBlock>
    <StackPanel HorizontalAlignment="Left">
        <TextBlock Margin="10" FontFamily="Arial" FontSize="20"
            FontWeight="Bold" Foreground="Orange">
            Desarrolla con
        </TextBlock>
        <Image Source="MSDN.gif" Height="50"/>
    </StackPanel>
```

```

<StackPanel Orientation="Horizontal">
    <Image Source="Windito.gif" Height="100"/>
    <TextBlock Margin="10" VerticalAlignment="Center" FontFamily="Arial"
        FontSize="20" FontWeight="Bold" Foreground="Navy">
        y aprende WPF con nosotros
    </TextBlock>
</StackPanel>
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
    <Button FontFamily="Arial" FontSize="20" FontWeight="Bold">
        Aceptar
    </Button>
    <Button FontFamily="Arial" FontSize="20" FontWeight="Bold">
        Cancelar
    </Button>
</StackPanel>
</StackPanel>
</Window>

```

Para cada manejador definido en el XAML hay que dar una implementación en el Codebehind como muestra el Listado 5-3. El nombre dado al manejador en el XAML tiene que coincidir con el nombre del método en la implementación en el Codebehind.

De manera general un manejador de evento tiene la siguiente sintaxis:

```
private void NombreDelManejador(object sender, RoutedEventArgs args){}
```

El parámetro sender expresa quién envía el evento directamente. Para obtener el elemento donde se originó un evento se tiene la propiedad Source del parámetro args de tipo RoutedEventArgs.

Los manejadores para los eventos relacionados con el ratón (MouseMove, MouseEnter o MouseLeave por ejemplo) son de tipo MouseEventHandler y su segundo parámetro es de tipo MouseEventArgs.

Para conocer las coordenadas del ratón el parámetro args brinda el método GetPosition que recibe como parámetros el elemento cuya esquina superior izquierda se tomará como origen de coordenadas para dar la ubicación relativa a ésta del puntero del ratón.

Listado 5- 3 Codebehind para el manejador de evento MouseMove

```
private void WindowMouseMove(object sender, MouseEventArgs
```

```

args){
    TB.Text = "(" + args.GetPosition(this).X + "," +
        args.GetPosition(this).X + ") : " + args.Source.GetType().Name;
}

```

Lo que hace este manejador (Listado 5-3) es mostrar la posición (coordenadas x,y) del cursor del ratón relativas a la ventana y el tipo del elemento sobre el que se encuentra (Figura 5-2) .



Figura 5 - 2 Manejando el MouseMove en la ventana

Existen situaciones en las cuales no resulta conveniente que el evento continúe la ruta, de hecho esto es lo que hace de modo predeterminado el control Button de WPF al convertir los eventos MouseButtonUp y MouseButtonUp en un evento Click, porque al elemento contenedor del botón en todo caso solo le interesa el saber que se hizo click sobre el botón.

Para detener la propagación en cualquier dirección de un Evento Enrutado se debe en un manejador de dicho evento dar valor true a la propiedad Handled del parámetro de tipo RoutedEventArgs. Esto provocará que el evento no continúe su ruta por el árbol.

5.2 Comandos en controles WPF

Un comando puede definirse como la forma de ejecutar una determinada acción. La mayoría de las opciones de menú con que contamos en las aplicaciones ejecutan un comando. Por ejemplo escoger la opción de menú **Edit** y luego la opción **Copy**, o hacer directamente **Ctrl+c**, son dos formas diferentes de invocar el comando **Copy**.

WPF brinda soporte para definir en XAML comandos que ejecuten un conjunto de acciones. La manera más sencilla de usar un comando en WPF se basa en un conjunto de RoutedCommand predefinidos, usar controles WPF que ya vengan con soporte para manejar el comando, y controles que tengan soporte para invocar el comando. Evitando de esta forma la necesidad de escribir código en el code-behind. Por ejemplo el control TextBox, tiene soporte para manejar

los comandos **Cut**, **Copy** y **Paste**, y el control Button tiene soporte para invocar comandos a través de la propiedad Command.

Asociadas a la propiedad Command hay a su vez dos propiedades: CommandParameter y CommandTarget, que permiten definir sobre quién se desea aplicar la acción del comando, así como el parámetro para la ejecución del comando (en el caso en que el comando necesite parámetro). Por ejemplo, el comando NavigationCommands.GoToPage requiere como parámetro la página hacia la cual navegar.

Cuando se invoca un comando WPF dispara dos eventos: PreviewExecuted y Executed (que son Tunneling y Bubbling respectivamente). Los eventos PreviewExecuted y Executed buscarán en el árbol de elementos contenedores por un objeto que tenga definido como manejar el evento (un CommandBinding para el comando específico que lanzó el evento). CommandBinding es el mecanismo (se ve con más detalle en la Lección **Enlace a Datos**) mediante el cual se asocia el evento con una lógica que lleva a cabo.

Existen en WPF un conjunto de clases que nos brindan comandos comúnmente utilizados, esto implica que no necesitamos crear nuestro propio RoutedCommand para operaciones como **nuevo, abrir, cerrar, cortar, copiar, pegar**, etc.

Existe una amplia gama de comandos definidos en WPF para ejecutar acciones en diversos escenarios. Estos comandos de tipo RoutedCommand están agrupados como propiedades estáticas en cinco clases diferentes.

- ApplicationCommands: Comandos estándar para cualquier aplicación, por ejemplo Copy, Paste, Undo, Redo, Find, Open, SaveAs, PrintPreview, etc.
- ComponentCommands: Comandos frecuentemente usados por la interfaz de usuario que brindan algunos controles. MoveLeft, MoveToEnd, ScrollPageDown, Textselection, etc.
- MediaCommands: Comandos usados para multimedia, por ejemplo Play, Pause, NextTrack, IncreaseVolume, ToggleMicrophoneOnOff, etc.
- NavigationCommands: Comandos utilizados para navegar por la información de las páginas por ejemplo BrowseBack, GoToPage, NextPage, Refresh, Zoom, etc.
- EditingCommands Comandos usados para editar documentos, por ejemplo AlignCenter, ApplyFontSize, EnterPageBreak, ToggleBold, Bold, Italic, Alignment, etc.

Cada uno de estos comandos es un "singleton" (existe una única instancia del comando). Esto implica que si registramos un manejador para el evento de invocar un comando, éste se ejecutará cada vez que el comando sea invocado en cualquier lugar de nuestra aplicación. Si lo que queremos es que este manejador se ejecute solo cuando el comando es invocado en un contexto determinado tenemos entonces que establecer una relación entre **comando, manejador y ámbito** del elemento que será el destino de la acción definida por el comando

dentro de la interfaz de usuario, para ello se hace uso de un objeto CommandBinding como veremos más adelante en esta misma lección.

Toda la lógica que se sigue en WPF para el trabajo con comandos podría separarse en cuatro conceptos fundamentales:

- La acción a ejecutar
- El objeto que invoca al comando
- El objeto sobre el cual se ejecuta el comando
- El objeto que relaciona el comando con una lógica.

Lo más interesante en la arquitectura de los comandos de WPF, es las distintas formas que hay de asociar un "destino" y una funcionalidad al comando. Vamos a dividir ahora nuestra explicación en dos escenarios.

5.2.1 Asociando un comando a un elemento con soporte para manejarlo.

Si Ud. desea comenzar a trabajar con los comandos de WPF, quizás lo primero que le venga a la mente probar es una caja de texto donde poder Cortar, Copiar y Pegar contenido. Cómo ya mencionamos anteriormente el control TextBox brinda soporte intrínseco para estos comandos, veamos el código XAML del Listado 5-4.

Listado 5-4 Código XAML de ventana con un Control TextBox

```
<Window x:Class="ComandosEnTextBox.Window1"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Comandos Cut,Copy y Paste en TextBox"
    Height="300" Width="300" >
    <StackPanel>
        <TextBox Name="TextBox1" Margin="10" FontSize="20"/>
    </StackPanel>
</Window>
```

El resultado aparece en la Figura 5-3. Note que la opción de **Pegar (Paste)** en el menú de contexto que despliega el TextBox, como respuesta a la acción de clic derecho del ratón, aparece deshabilitada pues en el clipboard no hay nada "a pegar" (en este caso el método CanExecute para el comando Paste devuelve false).

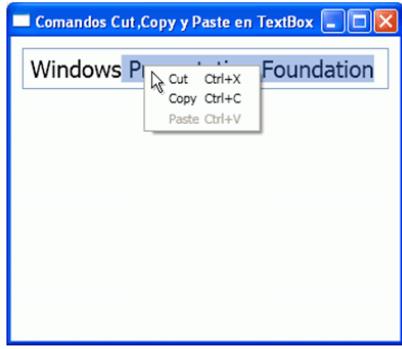


Figura 5 - 3 Soporte para manejar los comandos Cut, Copy y Paste en el TextBox

Ahora queremos que estas acciones de Cortar, Copiar y Pegar sobre la caja de texto se puedan lograr también a través de botones. Para ello consideremos el código del Listado 5-5

Listado 5- 5 Cortar Copiar y Pegar desde Botones

```
<StackPanel Margin="10">
    <StackPanel>
        <TextBox Name="TextBox1" FontSize="20"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal"
    HorizontalAlignment="Center">
        <Button Command="Cut" FontSize="18">Cortar</Button>
        <Button Command="Copy" FontSize="18">Copiar</Button>
        <Button Command="Paste" FontSize="18">Pegar</Button>
    </StackPanel>
</StackPanel>
```

La ejecución de este código desplegará la Figura 5-4.

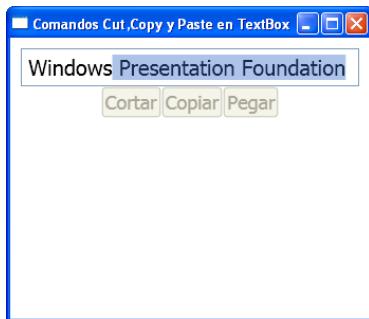


Figura 5 - 4 Botones Cortar, Copiar y Pegar deshabilitados

¡Pero los botones se muestran como deshabilitados! Note en la Figura 5-4 como aún seleccionando el texto del TextBox no se han habilitado ninguno de los botones con los que se

quiere invocar el comando. No se alarme. Es lógico que esto suceda, la razón es que en la ruta que siguen los eventos PreviewExecuted y Executed no existe un elemento que tenga definido como manejar estos comandos (Note que la caja de texto no se encuentra en la ruta). Y no se ha asociado explícitamente un elemento sobre el que ejecutará el comando. Cuando se asocia un comando a un control, este se deshabilita si el comando asociado no puede ejecutarse

Una solución para que los botones aparezcan como habilitados, es definir un CommandTarget para el comando, es decir un "destinatario" sobre el que actuará el comando. En el Listado 5-6 se ha escogido la caja de texto de nombre TextBox1 como destinatario de los comandos Cut, Copy y Paste.

Listado 5- 6 Dando valor a la propiedad CommandTarget	
<StackPanel Margin="10">	
<StackPanel>	
<TextBox Name="TextBox1" FontSize="20"/>	
</StackPanel>	
<StackPanel Orientation="Horizontal"	
HorizontalAlignment="Center">	
<Button Command="Cut"	
FontSize="18" CommandTarget="{Binding ElementName=	
TextBox1}">	
Cortar	
</Button>	
<Button Command="Copy"	
FontSize="18"	
CommandTarget="{Binding ElementName=TextBox1}">	
Copiar	
</Button>	
<Button Command="Paste"	
FontSize="18"	
CommandTarget="{Binding ElementName=TextBox1}">	
Pegar	
</Button>	
</StackPanel>	
</StackPanel>	



Figura 5 - 5 Botones Cortar, Copiar y Pegar habilitados

Possiblemente Ud se pregunte qué pasará ahora si el ComandTarget es uno solo y queremos tener más de una caja de texto de la cual quisiéramos Cortar, Copiar o Pegar. La respuesta es que solo podremos aplicar los comandos sobre el control que se ha vinculado al ComandTarget, lo cual podría ser una limitación. No se aflija, para resolver este problema usaremos la propiedad IsFocusScope de la clase FocusManager.

Cuando no se especifica explícitamente un valor a la propiedad CommandTarget, esta tomará como valor el elemento que tenga el foco. Pero en este caso no es posible mantener el foco sobre la caja de texto del cual se quiere Cortar, Copiar o Pegar y a la vez dar clic sobre uno de los botones (pues para dar clic el foco lo toma el botón).

Sin embargo con la propiedad IsFocusScope de tipo bool podemos especificar ámbitos de foco para un control. El código del Listado 5-7 nos ilustra cómo lograr esto. Se le ha dado un ámbito de foco al StackPanel que contiene a los tres botones de modo que el StackPanel y los TextBox estarán ahora en ámbitos de foco distintos. Note ahora en el Listado 5-7 como prescindimos de la propiedad CommandTarget pues automáticamente esta tomará como valor el elemento que contenga el foco y si además este tiene la capacidad de manejar el comando (como es el caso de los TextBox para los comandos Cut, Copy y Paste) pues ya todo estará listo.

Listado 5-7 Definiendo un ámbito de foco

```
<StackPanel Margin="10">
  <StackPanel>
    <TextBox Name="TextBox1" FontSize="20"/>
    <TextBox Name="TextBox2" FontSize="20"/>
  </StackPanel>
  <StackPanel FocusManager.IsFocusScope="True"
             Orientation="Horizontal"
             HorizontalAlignment="Center"
             Background="Red" Margin="10">
    <Button Command="Cut" FontSize="18">
      Cortar
    </Button>
  </StackPanel>
</StackPanel>
```

```

</Button>
<Button Command="Copy" FontSize="18">
    Copiar
</Button>
<Button Command="Paste" FontSize="18">
    Pegar
</Button>
</StackPanel>
</StackPanel>

```

Distinga en el Listado 5-7 el StackPanel que incluye los tres botones

```

<StackPanel FocusManager.IsFocusScope="True"
            Orientation="Horizontal"
            HorizontalAlignment="Center"
            Background="Red" Margin="10">

```

Note que en la Figura 5-6 Los tres botones bordeados de rojo (color de fondo del StackPanel que los contiene) tienen su propio ámbito de foco (definido por su contenedor StackPanel), lo cual quiere decir que los comandos definidos en estos botones son aplicables en cualquier elemento fuera de su ámbito de foco, entiendase en este caso a ambos controles TextBox pues al no asignar valor a la propiedad CommandTarget esta tomará como valor el elemento que tenga el foco.

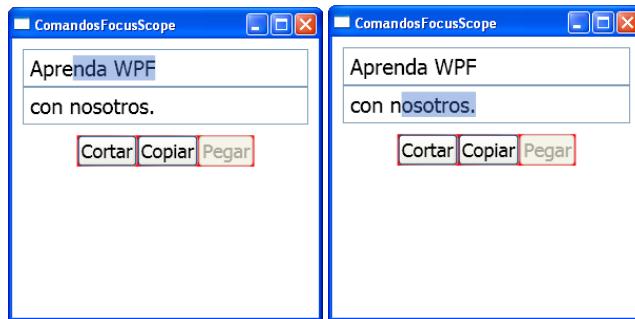


Figura 5 - 6 Botones Cortar, Copiar y Pegar habilitados

Por defecto los elementos de los controles Menu yToolBar tienen su propio ámbito de foco , así que los comandos definidos por controles dentro de ellos podrán funcionar en controles fuera de este ámbito sin necesidad de algún código adicional que lo especifique, pruebe y verá.

5.2.3 Asociando un comando a un elemento que no tiene soporte para manejarlo.

Hasta ahora hemos visto el desarrollo de un ejemplo en el cual el elemento sobre el que se aplica el comando conoce como manejarlo. Pero sería incorrecto terminar esta lección si planteárnos la siguiente interrogante ¿Qué pasa entonces si el elemento "destino" del comando no cuenta con soporte para el mismo?

Para entender esto mejor veamos por ejemplo como asociar el comando Close a un botón para cerrar una ventana cuando se haga clic sobre éste. La ventana (Window) no tiene definido como manejar el comando Close, por lo que habrá que asociar una lógica a la ejecución del comando mediante CommandBinding (Listado 5-8).

```
Listado 5- 8 Definiendo un CommandBinding
<Window x:Class="EjemploComandoClose.Window1"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="EjemploComandoClose" Height="300" Width="300" >
<Window.CommandBindings>
    <CommandBinding Command="Close"
        Executed="CloseCommandHandler"
        CanExecute="CloseCommandCanExecuteHandler"/>
</Window.CommandBindings>
    <StackPanel HorizontalAlignment="Center"
        VerticalAlignment="Center">
        <Button Height="50" Width="100" Command="Close">
            Cerrar Ventana
        </Button>
        <CheckBox Name="ChkB" IsChecked="True">
            Habilitar el comando Close
        </CheckBox>
    </StackPanel>
</Window>
```

Note como hemos definido en la ventana un objeto CommandBinding que relaciona el comando con una lógica determinada. Ahora hay que ir al code-behind y dar implementación a CloseCommandHandler y CloseCommandCanExecuteHandler manejadores que se han asociado al Executed y CanExecuted de CommandBinding.

Este es un ejemplo en que es necesario escribir código en el code-behind (Listado 5-9) pues de otra manera no habrá un elemento en la jerarquía que de funcionalidad al comando Close.

Listado 5- 9 code-behind para el comando Close

```
private void CloseCommandHandler(object sender,  
ExecutedRoutedEventArgs e)  
{  
    Close();  
}  
private void CloseCommandCanExecuteHandler(object sender,  
CanExecuteRoutedEventArgs e)  
{  
    if (ChkB != null)  
        if (ChkB.IsChecked.Value)  
            e.CanExecute = true;  
        else  
            e.CanExecute = false;  
}
```

La Figura 5-7 muestra el resultado de la ejecución. Note como se deshabilita el botón (Figura 5-8 derecha) si el método CloseCommandCanExecuteHandler retorna false debido a que no se ha seleccionado el CheckBox. Ahora Ud. puede "jugar" un poco con este código agregando condiciones para la ejecución del comando.



Figura 5 - 7 Comando Close desde Botón

Los eventos son un buen aporte en WPF para poder interactuar con los controles que forman la interfaz de una aplicación sin preocuparnos por la estructura de su contenido. Por su parte los comandos apoyados en los Eventos Enrutados, nos permiten ejecutar acciones gran parte de las cuales vienen ya predeterminadas y para las cuales no hay que implementar código alguno en el code-behind.

Lección 6 Enlace a Datos

En esta lección se verá cómo hacer que una interfaz de usuario refleje visualmente lo qué está pasando por detrás del telón con los datos de la aplicación, y a la inversa también podrá lograr que la interacción que realice con la interfaz pueda reflejarse en los datos de la aplicación. Veremos cómo esto se podrá hacer de forma sencilla en XAML.

6.1 Binding

La clase Binding, definida en el ensamblado PresentationFramework, dentro del espacio de nombres System.Windows.Data.Binding, representa a los objetos que permiten enlazar una fuente de datos con un destino de datos. Este concepto seguramente ya es conocido por el lector que haya trabajado con la plataforma .NET. El enlace de una fuente de datos con un destino de datos evita tener que escribir explícitamente código de la forma MiElementoDeInterfaz.Valor = MiAplicacion.Dato o el inverso MiAplicacion.Dato = MiElementoDeInterfaz.Valor a la vez que evita tener que programar la plomería necesaria para mantener sincronizados los datos en ambos extremos del enlace.

En la Figura 6-1 se ilustra el funcionamiento básico de la clase Binding. El objeto Binding está representado en la figura como la línea que une la fuente y el destino, en una o ambas direcciones. A su vez, la fuente y el destino pueden ser de diferentes proveedores de datos como gestores de datos, componentes de negocio o controles visuales que permiten la interacción con la aplicación.

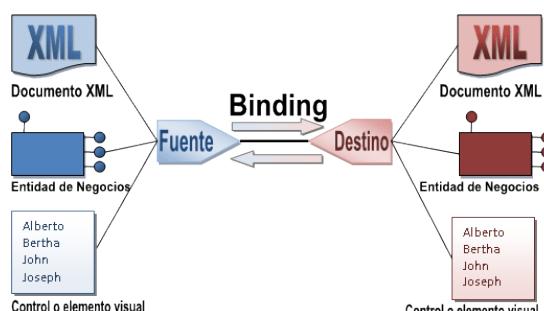


Figura 6-1 Esquema de uso de objetos de la clase Binding

Veamos con un ejemplo sencillo las capacidades básicas de esta clase. El Listado 6- 1 muestra un par de controles básicos, un TextBox y un TextBlock que están enlazados por un Binding.

Listado 6- 1 Controles que servirán de fuente y destino de datos al ejemplo
--

<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition/>

```

<ColumnDefinition/>
</Grid.ColumnDefinitions>
<TextBox Name="tbSource">Texto fuente</TextBox>
  <TextBlock Grid.Column="1" Text="{Binding ElementName=tbSource,
Path=Text}"/>
</Grid>

```

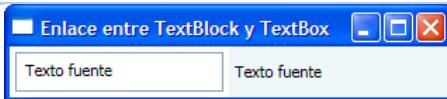


Figura 6- 2 A la izquierda un TextBox (fuente de datos) y a la derecha un TextBlock (destino de datos).

Para crear el enlace, hay que determinar los dos extremos del mismo. En el extremo destino es donde se ubica el objeto Binding para indicar con las propiedades del objeto Binding cuál es la fuente de los datos. En el listado se muestra el enlace entre ambos elementos. En este ejemplo se le ha dado un nombre a la fuente a través de la propiedad Name para poderla referir desde el objeto Binding. En el elemento destino, TextBlock en este ejemplo, se le asigna el objeto Binding a la propiedad con la que quiere enlazarse la fuente (la propiedad Text en este ejemplo).

Observe el uso de las llaves, en este caso {Binding ...} para XAML significa no tomar la cadena literalmente sino que se interprete esta cadena para resolver el enlace y determinar el valor que se le dará a la propiedad Text del TextBlock. La propiedad ElementName le indica al Binding que el objeto que sirve como fuente de datos tiene como nombre tbSource. La propiedad Path del Binding indica cuál propiedad del objeto fuente será utilizada para obtener el dato (en este caso la propiedad Text de tbSource).

La Figura 6- 3 muestra el resultado de desplegar el ejemplo con estas modificaciones, justo cuando la ventana es desplegada, y justo después de escribir la cadena "Bind me if you can" en la caja de texto.



Figura 6- 3 Al crear el enlace el dato se copia inmediatamente desde la fuente hacia el destino. Al variar el valor de la fuente, el enlace mantiene actualizado el destino.

A un enlace se le puede indicar el momento y la dirección en que debe mantener el enlace. Esto se logra mediante la propiedad Mode de la clase Binding que es de tipo BindingMode. Por

ejemplo, si en el Listado 6- 1 se hubiese escrito "{Binding ElementName=tbSource, Path=Text, Mode=OneTime}" significaría que el enlace solo se debe activar la primera vez al desplegar el TextBlock, pero si luego se modifica el texto del TextBox, veremos que nada cambia en el TextBlock.

Hay otros modos más útiles como son unidireccional (OneWay), unidireccional invertido (OneWayToSource) y bidireccional (TwoWay). Estos, como sus nombres indican, permiten activar el enlace solo cuando se actualiza la fuente, solo cuando se actualiza el destino, o cuando se actualiza cualquiera de los dos, respectivamente. Finalmente el valor Default permite decidir, al sistema de propiedades de WPF, el comportamiento del enlace. Este es el valor que toma la propiedad Mode del enlace cuando no se especifica un valor y que se traduce en la práctica en OneWay o TwoWay en dependencia de la propiedad destino del enlace.

Para la propiedad Text del TextBox el modo TwoWay es el que se asume cuando no se indica un valor para la propiedad Mode. Sin embargo para la propiedad Text del TextBlock, cuando hace de destino de un enlace, se asume el modo OneWay. La distinción se hace para las propiedades de controles que reflejan algún valor que se puede modificar desde la interfaz de usuario, como la propiedad Value de un Slider, o el Text de TextBox o el IsChecked de CheckBox. Estas propiedades, se asumen en modo bidireccional TwoWay por defecto, mientras que el resto de las propiedades como el Width del Slider, el FontSize del TextBox o el Content del CheckBox se asumen en modo unidireccional OneWay.

La Figura 6- 4 ilustra el comportamiento de los diferentes valores de la propiedad Mode en el uso de enlaces.

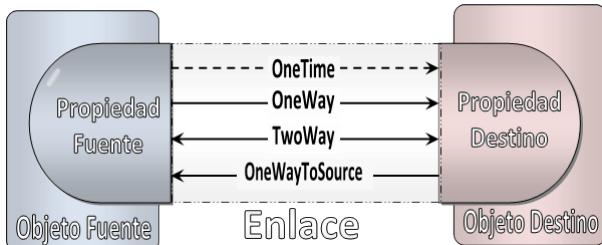


Figura 6- 4 Efectos del uso del modo de enlace.

6.2 Fuentes de datos

El objeto destino de datos de un enlace siempre es un DependencyObject (clase base de la mayoría de clases en WPF como es el caso de los elementos visuales y los controles que son los que se usan como destino en la mayoría de las aplicaciones) y la propiedad destino es una DependencyProperty de este objeto (la mayoría de las propiedades en WPF están en esta categoría). Esta restricción está respaldada por una implementación muy eficiente de la gestión del destino del enlace, por la forma en que están implementadas las clases en WPF.

Sin embargo, el objeto fuente de datos sí puede ser de muy diversa índole. Se puede tratar de un objeto de negocio, de un WebService, un control (DependencyObject), una colección de

objetos, o incluso ¡un documento XML! Para cada uno de estos casos, la propiedad que expresa la fuente de datos puede ser una simple propiedad en términos de .NET, una DependencyProperty, un elemento de una colección, un atributo de XML, ¡o hasta toda una ruta de navegación hasta la fuente de datos!

6.2.1 Un Objeto de Negocio como fuente de Datos

Veamos cómo hay que implementar un objeto de negocios de manera que se pueda utilizar como una fuente de datos.

El Listado 6- 2, nos muestra un ejemplo de una clase Film, con algunos datos como el Titulo, el Género, si es ganadora de un Oscar y la Calificación que otorga la crítica, un valor entre 0 y 5.

Listado 6- 2 Implementación de un objeto de negocios simple

```
public enum Genero
{
    Mafia, Drama, Catastrofismo, Aventuras, Comedia, Oeste,
    CienciaFiccion
}
public class Film {
    string titulo;
    Genero genero;
    bool? oscar;
    double calificación;
    public Film(){}
}
public Film(string titulo, Genero genero, bool? oscar, double calificación){
    this.Titulo = titulo;
    this.Genero = genero;
    this.Oscar = oscar;
    this.Calificación = calificación;
}
public string Titulo{
    get{ return titulo; }
    set{ titulo = value; }
}
public Genero Genero {
    get{ return genero; }
    set{ genero = value; }
}
public bool? Oscar {
    get{ return oscar; }
    set{ oscar = value; }
}
```

```

public double Calificación {
    get{ return calificación; }
    set{ calificación = value; }
}
}

```

Para poder crear un objeto de negocio desde el código XAML la clase debe tener un constructor por defecto, de este modo en el código XAML se pueden indicar cuáles propiedades se inicializan. Esto es lo que ocurre en el Listado 6- 3 cuando se hace

```

<Window.Resources>
<ejemplo:Film x:Key="filmInfiltrados" Titulo="Infiltrados" Genero="Mafia"
Oscar="{x:Null}" Calificación="4.7"/>
</Window.Resources>

```

Si aún no está familiarizado con .NET 2.0 sepia que un tipo T? es un tipo que también incluye el valor null La notación {x:Null} permite en XAML representar al valor null, es decir en este caso un valor que se desconoce en el momento de escribir el código.

Si Usted es cinéfilo y en el momento de estudiar este curso aún no se ha hecho la premiación de los Oscar del 2007 sígale la pista a este film porque es un posible ganador

Listado 6- 3 Enlace en XAML con un objeto de negocios para visualizarlo, editarlo y modificarlo

```

<Window x:Class="WPF_EnlaceDatos.EnlaceObjetoNegocio"
xmlns:ejemplo="clr-namespace:WPF_EnlaceDatos"
...
>
<Window.Resources>
...
<ejemplo:Film x:Key="filmInfiltrados" Titulo="Infiltrados"
Genero="Mafia" Oscar="{x:Null}" Calificación="4.7"/>
</Window.Resources>
<Grid ...>
...
<!-- Visualización del objeto de negocios -->
<Grid ...>
...
<TextBlock Text="{Binding Path=Titulo,
Source={StaticResource filmInfiltrados}}"
FontWeight="Bold" FontSize="16"/>
<TextBlock Text="Género:</TextBlock>
<TextBlock Text="{Binding Genero,
Source={StaticResource
filmInfiltrados}}"/>

```

```

<TextBlock Text="Ganó Oscar:"/>
<TextBlock Text="{Binding Oscar, Source={StaticResource filmInfiltrados}}"/>
<TextBlock Text="Calificación: " />
<TextBlock Text="{Binding Calificación,
    Source={StaticResource filmInfiltrados}}"/>
</Grid>
<!-- Edición del objeto de negocios --&gt;
&lt;Grid ...&gt;
...
&lt;TextBox Text="{Binding Titulo, Source={StaticResource filmInfiltrados}}"
    FontWeight="Bold" FontSize="16"/&gt;
&lt;TextBlock Text="Género:"/&gt;
&lt;TextBox Text="{Binding Genero, Source={StaticResource filmInfiltrados}}"/&gt;
&lt;TextBlock Text="Ganó Oscar: " /&gt;
&lt;TextBox Text="{Binding Oscar, Source={StaticResource filmInfiltrados}}"/&gt;
&lt;TextBlock Text="Unidades: " /&gt;
&lt;TextBox Text="{Binding Calificación,
    Source={StaticResource filmInfiltrados}}"/&gt;
&lt;/Grid&gt;
&lt;/Grid&gt;
&lt;/Window&gt;
</pre>

```

Este código nos enlaza como destino propiedades de dos elementos visuales (un TextBlock y un TextBox) con propiedades fuente de un mismo objeto de negocio de tipo Film que ha sido guardado en los recursos con la llave filmInfiltrados.

A la hora de definir un enlace, la propiedad más importante es generalmente Path. Cuando se escribe solamente {Binding <nombre de propiedad>}, se entiende como una forma abreviada de escribir {Binding Path=<nombre de propiedad>, ...}.

Este código muestra cómo crear los enlaces deseados entre los controles y las propiedades correspondientes del objeto. En el Listado 6-1 se especificó la fuente poniendo el ElementName y el Path para indicar la propiedad para el enlace, pero esto es válido solo cuando la fuente de datos es un elemento que está ubicado dentro del XAML. Pero en este caso la fuente es un objeto de tipo Film que no es resultado de una definición explícita de un elemento en XAML. Se usa entonces la propiedad Source del Binding, para indicar cuál es el objeto fuente de datos. El valor de Source es aquí un {StaticResource filmInfiltrados} que indica que se debe acceder al valor que se ha puesto en el diccionario de recursos con la llave filmInfiltrados.. El valor asignado a la propiedad Path (que en Listado 6-4 se denota de forma abreviada poniendo solo el valor) nos indica el nombre de la propiedad con la que dentro de este objeto fuente se quiere hacer el enlace.

Si prueba a ejecutar el ejemplo tal y como está hasta este punto, notará cómo se visualizan los datos del Film en el TextBlock y en el TextBox de la Figura 6- 5.

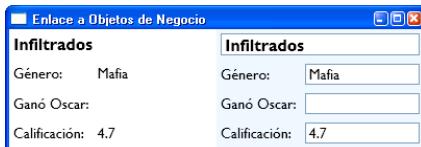


Figura 6- 5 Interfaz para visualizar y modificar un objeto de negocio

Si modificamos interactivamente los textos en las cajas de texto los cambios se reflejan en las propiedades correspondientes del TextBlock. Esto se debe a que la propiedad Text de la caja de texto esta ligada TwoWay con la fuente que es la propiedad en el objeto de negocios (objeto Film en este caso) que a su vez está ligada OneWay con la propiedad Text de los TextBlock. Todo esto lo lleva a cabo la maquinaria de WPF.

Sin embargo, si el valor de las propiedades del objeto de negocio es cambiado directamente dentro de la lógica de la aplicación este cambio no se refleja en los elementos visuales enlazados con esa propiedad.

Es lógico esperar que así sea porque no es posible saber desde el código .NET cuándo se cambia una propiedad si en el código de esta no se notifica de alguna manera el cambio. Recuerde que incluso desde el código .NET una propiedad pudiera cambiarse sin que se haya hecho a través de el código de una parte set.

Para lograr que un cambio en una propiedad de un objeto de negocio, producido desde el código .NET, se refleje en un elemento visual ligado a esta propiedad debe programarse este código .NET siguiendo un patrón como el que se describe a continuación.

Notificación de cambio de valor de una propiedad

Todo objeto de negocio que quiera notificar los cambios en el valor de sus propiedades debe implementar la interfaz INotifyPropertyChanged (Listado 6- 4) y seguir un patrón como el que se muestra en el Listado 6- 5

Listado 6- 4 definición de la interfaz INotifyProperty

```
public interface INotifyPropertyChanged {
    event PropertyChangedEventHandler
        PropertyChanged;
}
```

Note como en la parte set de cada propiedad se ha llamado al método OnPropertyChanged que a su vez se encarga de notificar el evento PropertyChanged. Con esta notificación WPF se encarga de actualizar los elementos visuales que están enlazados con el objeto (parámetro this) que ha disparado el evento.

Listado 6- 5 Implementación de la interfaz INotifyPropertyChanged por un objeto de negocios

```
using System.ComponentModel;
namespace WPF_EnlaceDatos {
    public class Notificador : INotifyPropertyChanged {
        public event PropertyChangedEventHandler PropertyChanged;
        protected void OnPropertyChanged(string name){
            if (PropertyChanged != null)
                PropertyChanged(this, new
PropertyChangeEventArgs(name));
        }
        public enum Genero { Mafia, Drama, Catastrofismo, Aventuras,
Comedia, Oeste, CienciaFiccion }
        public class Film : Notificador {
            string titulo;
            Genero genero;
            bool? oscar;
            double calificación;
            public Film(){}
            public Film(string titulo, Genero genero, bool? oscar, double
calificación){
                this.Titulo = titulo;
                this.Genero = genero;
                this.Oscar = oscar;
                this.Calificación = calificación;
            }
            public string Titulo{
                get{ return titulo; }
                set{
                    if (titulo == value) return;
                    titulo = value;
                    OnPropertyChanged("Titulo");
                }
            }
            public Genero Genero {
                get{ return genero; }
                set{
                    if (genero == value) return;
                    genero = value;
                    OnPropertyChanged("Genero");
                }
            }
        }
    }
}
```

```
        }
    }

    public bool? Oscar {
        get{ return oscar; }
        set{
            if (oscar == value) return;
            oscar = value;
            OnPropertyChanged("Oscar");
        }
    }

    public double Calificación {
        get{ return calificación; }
        set{
            if (calificación == value) return;
            calificación = value;
            OnPropertyChanged("Calificación");
        }
    }
}
```

Aunque no es necesario hacer la clase Notificador esta nos servirá para heredar el método OnPropertyChanged en los objetos de negocios. Es perfectamente posible mover este método y el evento PropertyChanged hacia la clase Film.

Cuando se hace una invocación del tipo `PropertyChanged(...)` se le informa a WPF cuál propiedad ha cambiado para que éste se encargue de reflejar el cambio en las propiedades destino que estén enlazadas con ésta, como seguramente las propiedades destino son renderizables (como lo son las de este ejemplo) entonces el cambio se visualizará en la interfaz.

Podemos ahora colocar un botón al Listado 6- 6 y obtener el Listado 6- 7 para que el manejador del clic del botón cambie en el code-behind los valores de las propiedades del objeto de negocio. El resultado es el que se muestra en la Figura 6- 5

Listado 6- 6 Enlace en XAML con un objeto de negocios para visualizarlo, editarlo y modificarlo

```
<Window x:Class="WPF_EnlaceDatos.EnlaceObjetoNegocio"
    xmlns:ejemplo="clr-namespace:WPF_EnlaceDatos"
    ...
    >
<Window.Resources>
    ...
    <ejemplo:Film x:Key="filmInfiltrados" Titulo="Infiltrados"
        Genero="Mafia" Oscar="{x:Null}" Calificación="4.7"/>
</Window.Resources>
<Grid ...>
    ...
    <!-- Visualización del objeto de negocios -->
    <Grid ...>
        ...
        <TextBlock Text="{Binding Path=Titulo,
            Source={StaticResource filmInfiltrados}}"
            FontWeight="Bold" FontSize="16"/>
        ...
    </Grid>
    <!-- Edición del objeto de negocios -->
    <Grid ...>
        ...
        <TextBox Text="{Binding Titulo, Source={StaticResource filmInfiltrados}}"
            FontWeight="Bold" FontSize="16"/>
        ...
    </Grid>
    <StackPanel>
        <Button Click="Modify1_Click">Cambiar por Titanic</Button>
    </StackPanel>
</Grid>
</Window>
```

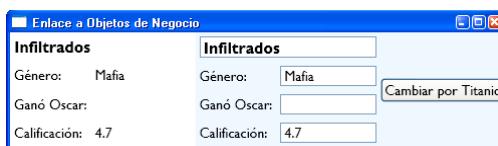


Figura 6- 6 El botón permite modificar los datos del objeto desde código C# sin la participación de enlaces.

El código del manejador Modify1_Click del evento Click del botón se muestra en el Listado 6-8. Ahora cuando se cambian los valores de las propiedades los set correspondientes de cada una notificarán a WPF del cambio que se encargará de reflejarlo en los elementos visuales enlazados a ella mostrándose el resultado como en la Figura 6-7

Listado 6- 7 Modificando un objeto de negocios desde C#

```
void Modify1_Click(object sender, RoutedEventArgs args) {
    Film film = Resources["filmInfiltrados"] as Film;
    film.Titulo = "Titanic";
    film.Genero = Genero.Catastrofismo;
    film.Oscar = true;
    film.Calificación = 4.2;
}
```



Figura 6- 7 El patrón de uso de la interfaz **IPropertyChanged** permite reflejar en la interfaz de usuario los cambio en la lógica de negocio

Rutas Complejas

Si una aplicación utiliza varias clases de objetos de negocio, es probable que exista alguna relación entre estos, como ocurre por ejemplo entre un objeto Film y un objeto Director si el Film tiene una referencia a su Director (Listado 6- 8). También es posible que existan relaciones con multiplicidad mayor que uno, como la que existe entre un Film y los Actores. Usted puede completar esta definición para hacerla implementar la interfaz **IPropertyChanged** del mismo modo que se ha visto en la sección anterior.

Listado 6- 8 Clase de negocio Director y relaciones entre esta y Film

```
public class Film : Notificador {
    ...
    public string Titulo { ... }
    public Director Director { ... }
    public double Calificación { ... }
    public bool? Oscar { ... }
    public Genero GeneroPrincipal { ... }
    public ObservableCollection<string> Actores { get {...} }
```

```

...
}
public class Director : Notificador {
...
    public string Nombre { ... }
    public string Nacionalidad { ... }
    public ObservableCollection<string> Filmes { get {...} }
...
}

```

El código del Listado 6- 9 declara una instancia de Film incluyendo una instancia de Director como valor de la propiedad Director de Film.

Listado 6- 9 Objeto de tipo Film declarado en los recursos

```

<Window.Resources>
<ejemplo:Film x:Key="filmInfiltrados" Titulo="Infiltrados" Genero="Mafia"
    Oscar="{x:Null}" Calificación="4.7">
<ejemplo:Film.Director>
    <ejemplo:Director Nombre="Martin Scorsese" Nacionalidad="USA"/>
</ejemplo:Film.Director>
<ejemplo:Film.Actores>
    <system:String>Leonardo Di Caprio</system:String>
    <system:String>Jack Nicholson</system:String>
    <system:String>Mat Dammon</system:String>
    <system:String>Martin Scheen</system:String>
    <system:String>Alec Baldwin</system:String>
</ejemplo:Film.Actores>
</ejemplo:Film>
</Window.Resources>

```

Supongamos que queremos visualizar los datos del Film acompañados de algunos de los datos del Director, y el nombre del primer Actor, al estilo de la Figura 6- 8.

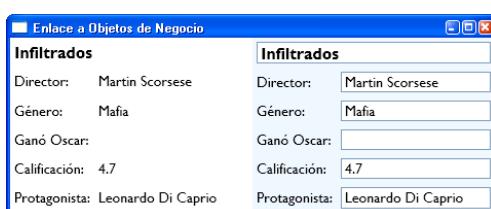


Figura 6- 8 Visualización de los datos de un film y su director usando enlaces

En el Listado 6- 10 se describen los fragmentos de XAML necesarios para añadir el nombre del director y del primer protagonista como en la Figura 6- 8.

Listado 6- 10 Uso de rutas complejas para acceder a datos que se pueden acceder indirectamente desde la fuente de datos.

```
<Window x:Class="WPF_EnlaceDatos.EnlaceObjetoNegocio"
    xmlns:ejemplo="clr-namespace:WPF_EnlaceDatos"
    xmlns:system="clr-namespace:System;assembly=mscorlib" >
<Window.Resources>
    <ejemplo:Film x:Key="filmInfiltrados" Titulo="Infiltrados" Genero="Mafia"
        Oscar="{x:Null}" Calificación="4.7">
        <ejemplo:Film.Director>
            <ejemplo:Director Nombre="Martin Scorsese" Nacionalidad="USA"/>
        </ejemplo:Film.Director>
        <ejemplo:Film.Actores>
            <system:String>Leonardo Di Caprio</system:String>
            <system:String>Jack Nicholson</system:String>
            <system:String>Mat Dammon</system:String>
            <system:String>Martin Scheen</system:String>
            <system:String>Alec Baldwin</system:String>
        </ejemplo:Film.Actores>
    </ejemplo:Film>
</Window.Resources>
<Grid>
    ...
    <!-- Visualización del objeto de negocios -->
    <Grid Grid.Row="0" Grid.Column="0">
        ...
        <TextBlock Text="{Binding Titulo, Source={StaticResource filmInfiltrados}}"
            FontWeight="Bold" FontSize="16" Grid.ColumnSpan="2"/>
        <TextBlock Text="Director:" Grid.Row="1" Grid.Column="0"/>
        <TextBlock Text="{Binding Director.Nombre,
            Source={StaticResource filmInfiltrados}}" Grid.Row="1" Grid.Column="1"/>
        ...
        <TextBlock Text="Protagonista:" Grid.Row="5" Grid.Column="0"/>
        <TextBlock Text="{Binding Actores[0],
            Source={StaticResource filmInfiltrados}}" Grid.Row="5" Grid.Column="1"/>
    </Grid>
    <!-- Edición del objeto de negocios -->
    <Grid Grid.Row="0" Grid.Column="1">
        ...
        <TextBox Text="{Binding Titulo, Source={StaticResource filmInfiltrados}}"
            FontWeight="Bold" FontSize="16" Grid.ColumnSpan="2"/>
        <TextBlock Text="Director:" Grid.Row="1" Grid.Column="0"/>
        <TextBox Text="{Binding Director.Nombre,
            Source={StaticResource filmInfiltrados}}" Grid.Row="1" Grid.Column="1"/>
```

```

    Grid.Column="1"/>
    ...
    <TextBlock Text="Protagonista:" Grid.Row="5" Grid.Column="0"/>
        <TextBox Text="{Binding Actores[0], Source={StaticResource
    filmInfiltrados}}"
            Grid.Row="5" Grid.Column="1"/>
    </Grid>
</Grid>
</Window>

```

Note la ruta utilizada Director.Nombre, esto indica que se está tomando al objeto filmInfiltrados como fuente de datos y a partir de ahí, se sigue la notación punto (dot notation) Director.Nombre para indicar la fuente original del dato..

El uso de rutas complejas en los enlaces, nos permite también acceder a datos "indizados" por enteros (bien porque es un array o porque tienen un indexer), como en el enlace (señalado en negritas) Actores[0], con el que se obtiene el primer actor del Film.

Que nos disculpen los cinéfilos, sabemos que en un film puede tener más de un protagonista pero hemos mostrado solo uno de ellos para ganar en brevedad.

Estas dos capacidades se pueden combinar para obtener rutas más complejas aún. Por ejemplo si la fuente de datos es un Director, las rutas Filmes[0].Actores[0] y Filmes[3].Titulo son correctas y permiten extraer, respectivamente, el primer protagonista del primer film del director, y el título del cuarto film.

6.2.2 Contexto de Datos

Todos los ejemplos que se han visto hasta ahora en esta lección se refieren explícitamente a un objeto que hace de fuente de datos (a través de las propiedades ElementName o Source) y una propiedad o ruta (Path o XPath) que sirve como fuente directa a los datos dentro de este objeto. Este patrón, si bien es suficiente para casos sencillos, carece de una característica que introduciremos en este epígrafe.

La propiedad RelativeSource es otra manera de obtener una fuente de datos, principalmente cuando se trata de plantillas de controles. Esta puede tomar los valores Self, TemplatedParent, PreviousData y FindAncestor, todos ellos permiten ubicar la fuente a partir del elemento al que se le aplica la plantilla.

El que se indique explícitamente la fuente de datos en cada enlace, hace difícil cambiar la misma dentro del propio XAML porque habría que estar localizando y cambiando los objetos Binding a lo largo de todo el código. Mas difícil aún sería si la fuente de datos cambia en tiempo de ejecución.

Sería diferente si se pudiera expresar una fuente de datos en función de algún elemento de referencia, y luego, en los enlaces que se declaren dentro del elemento, todas las rutas se tomasen como relativas a esta fuente. De este modo un cambio en el elemento de referencia común, significaría un cambio de la fuente de datos para todos los enlaces declarados en este elemento. Esto es lo que se hace en el Listado 6- 11. A la propiedad DataContext definida en FrameworkElement se le asocia el objeto fuente de datos y es el valor de esta propiedad el que será tomado como punto de partida para las rutas de enlaces que aparezcan dentro del FrameworkElement (y que no especifiquen explícitamente otra una fuente de datos).

La primera línea del Listado 6- 11 define un panel Grid que contiene varios elementos. La propiedad DataContext del Grid se “enlaza” con el recurso filmInfiltrados. Note que además ahora se escribe mas abreviadamente Text="{Binding Titulo}" en lugar de como hacíamos antes Text="{Binding Titulo, Source={StaticResource filmInfiltrados}}".

Listado 6- 11 Elemento con DataContext

```
<!-- Visualización del objeto de negocios -->
<Grid           DataContext="{StaticResource
filmInfiltrados}">
    <TextBlock Text="{Binding Titulo}" />
    <TextBlock Text="{Binding Director.Nombre}" />
    <TextBlock Text="{Binding Genero}" />
    <TextBlock Text="{Binding Oscar}" />
    <TextBlock Text="{Binding Calidicación}" />
    <TextBlock Text="{Binding Actores[0]}" />
</Grid>
```

Cuando se ha omitido la fuente de datos al declarar un enlace, WPF busca si en el elemento donde se está haciendo el enlace la propiedad DataContext tiene un valor diferente de null, en tal caso toma este valor como la fuente de datos. Si el valor es null, se sigue al padre del elemento (es decir al elemento que lo contiene) y así sucesivamente hasta encontrar un elemento que tenga definida esta propiedad, como ha sido el caso del Grid en este ejemplo. Otra ventaja que brinda este enfoque es que si se cambia su valor durante la ejecución, los enlaces que utilicen el valor de esta propiedad se adaptarán al cambio.

6.2.3 Enlace con XML y Proveedores de Datos

La clase Binding está preparada también para indicar que la fuente de datos es un documento XML.

Para una fuente de datos más compleja, como es el caso de un documento XML, no es suficiente el uso de rutas como se ha hecho en las secciones anteriores. En esos casos WPF

utiliza la capacidad de reflexión de .NET para conocer, según los nombres de las propiedades, la validez de la ruta.

Cuando la fuente de datos no es un objeto de negocios .NET sino una fuente XML, en lugar de Path, se debe utilizar la propiedad XPath. Un aspecto a tener en cuenta en este caso es dónde está la fuente. Para ello utilizaremos a la clase XmlDataProvider, que representa una consulta a un documento XML. Por ejemplo, el Listado 6- 12 muestra una instancia de XmlDataProvider que hace de fuente de datos XML, esta instancia se ha ubicado en el diccionario de recursos con la llave "filmInfiltrados". Para simplificar hemos puesto aquí el XML embebido dentro del elemento <x:XData>. Realmente es poco común que en tiempo de diseño se disponga de los datos para ponerlos en XML explícitamente dentro del código, lo usual es que la fuente de datos sea un documento XML que no tiene por qué conocerse en tiempo de diseño, más adelante veremos cómo se establece en este caso la asociación a la fuente.

Listado 6- 12 Segmento de código XML que hace de fuente de datos

```
...
<XmlDataProvider x:Key="film1" XPath="Film">
  <x:XData>
    <Film xmlns="" Titulo="Infiltrados" Genero="Mafia"
      Oscar="{x:Null}" Calificación="4.7">
      <Director Nombre="Martin Scorsese" Nacionalidad="USA"/>
      <Actores>
        <Actor Nombre="Leonardo Di Caprio"/>
        <Actor Nombre="Jack Nicholson"/>
        <Actor Nombre="Mat Dammon"/>
        <Actor Nombre="Martin Scheen"/>
        <Actor Nombre="Alec Baldwin"/>
      </Actores>
    </Film>
  </x:XData>
</XmlDataProvider>
...
```

Un elemento importante en esta forma de disponer de una fuente de datos XML es el uso del atributo xmlns, cuyo valor debe coincidir con el URI del esquema de datos del XML embebido, o debe ser cadena vacía si no hay una URI asociado al esquema. Este atributo anula el uso del atributo homónimo en la raíz del documento XAML

(xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation")

el cual declara que todos los elementos sin prefijo en el documento, pertenecen al esquema de presentación de XAML.

Para acceder a los datos de una fuente XML hay que conocer el lenguaje XPath, lo que va más allá de los objetivos de este curso. No obstante para que tenga una idea general le ilustraremos algunos ejemplos.

Con la indicación `XPath="Film"` en el Listado 6- 12 se especifica que del objeto `XmlDataProvider` solo se va a tomar como fuente de datos los elementos del documento que aparezcan a primer nivel (recuerde que la estructura de XML es anidada) de nombre Film. Si de este XML se fuese a tomar como fuente de datos solamente la lista de actores, se debería hacer entonces `XPath="Film/Actores"` o `XPath="//Actores"`.

Para enlazar los datos del documento XML y visualizarlos en la interfaz de usuario se procede como en el Listado 6- 13 lo que nos da el resultado de la Figura 6-9

Listado 6- 13 Enlaces a una fuente de datos XML, con el uso del lenguaje XPath

```
<StackPanel DataContext="{StaticResource film1}">
    <TextBlock Text="{Binding XPath=@Titulo}" FontWeight="Bold" FontSize="16"/>
    <TextBlock>
        <TextBlock Text="Director: " Margin="0"/>
        <TextBlock Text="{Binding XPath=Director/@Nombre}" FontWeight="Bold" Margin="0"/>
    </TextBlock>
    <TextBlock>
        <TextBlock Text="El protagonista principal es" Margin="0"/>
        <TextBlock Text="{Binding XPath=/Actor[1]/@Nombre}" Margin="0"/>
    </TextBlock>
</StackPanel>
```

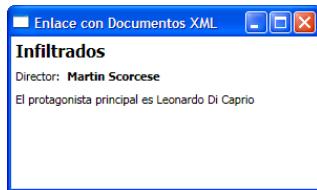


Figura 6- 9 Enlace con datos de un documento XML

Aunque semánticamente estas rutas de XPath son similares a sus rutas homólogas escritas en algún lenguaje de .NET, el lenguaje XPath tiene posibilidades de navegación que no se pueden lograr con la reflexión de .NET. Por ejemplo en .NET no sería posible navegar desde un objeto Film hacia su Director si la clase Film no tuviese una propiedad Director. Sin embargo, por la estructura anidada arbórea de XML, en XPath es posible pasar de cualquier elemento hijo a su correspondiente parente con la ruta "..", por lo que como en el Listado 6- 12 Director está dentro de Film es posible entonces navegar hasta el título del film con la ruta "../@Nombre".

Por otra parte, también es posible en XPath utilizar una semántica más compleja como usar predicados para extraer determinadas partes del documento que cumplan con cierta condición. Por ejemplo, la ruta "`//Actor[2]/@Nombre`" , aquí el predicado [2] es una forma abreviada de decir `[position()=2]`. Otros enlaces permitirían aplicar una condición más sofisticada sobre los atributos de los elementos, como "`//Film[@Calificación>=2]/@Titulo`" que extrae la lista de los títulos de los filmes cuya calificación es mayor que 2, en el supuesto caso en que el documento XML contuviese varios filmes.

Otra manera de indicar una fuente de datos XML es utilizando los recursos de la aplicación. Para ello se debe incluir en el proyecto un archivo XML como se muestra en la Figura 6- 10 y asegurarse de que en el tipo de acción en las propiedades del documento se haya indicado Resources para que Visual Studio lo incluya en los recursos de la DLL.

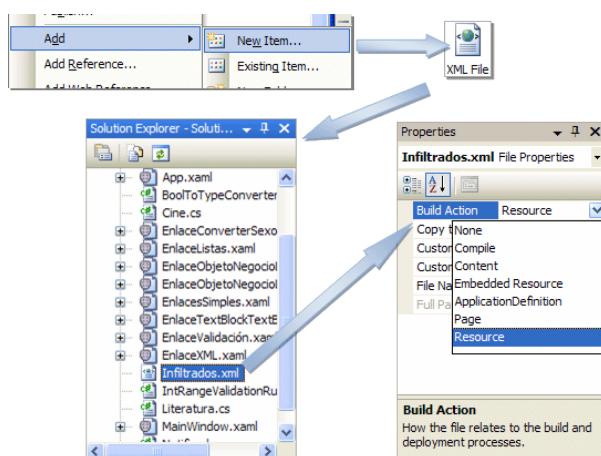


Figura 6- 10 Creación de un documento XML en los recursos para utilizar desde XAML

Con esto podemos utilizar en XAML la propiedad `Source` de `XmlDataProvider`, como en el Listado 6- 14. El efecto es el mismo en los ejemplos anteriores, con la diferencia que si se utiliza el mismo documento en varias partes de la aplicación, no hay que replicar los datos en cada página.

Listado 6- 14 Uso de un documento ubicado en los recursos de la aplicación

```
<XmlDataProvider x:Key="film1" XPath="Film" Source="Infiltrados.xml"/>
```

El contenido del archivo `Infiltrados.xml` se muestra en el Listado 6- 15.

Listado 6- 15 Contenido del archivo Infiltrados.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Film Titulo="Infiltrados" Genero="Mafia" Oscar="{x:Null}" Calificación="4.7">
<Director Nombre="Martin Scorsese" Nacionalidad="USA"/>
```

```

<Actores>
  <Actor Nombre="Leonardo Di Caprio"/>
  <Actor Nombre="Jack Nicholson"/>
  <Actor Nombre="Mat Dammon"/>
  <Actor Nombre="Martin Scheen"/>
  <Actor Nombre="Alec Baldwin"/>
</Actores>
</Film>

```

Otra forma de disponer de fuente de datos XML es utilizando la propiedad Document, de tipo XmlDocument, de XmlDataProvider, con esta propiedad se indica la dirección de dónde cargar desde el sistema de archivos o de la red. De esta forma se puede lograr más dinamismo variando el valor de Source podemos lograr que cambie la fuente, o modificando el valor de la ruta XPath, puede provocar que los elementos extraídos del documento sean diferentes.

6.3 Colecciones como fuentes de datos

Una fuente de datos no siempre es un solo elemento, sino que puede ser toda una colección de elementos. Por ejemplo puede ser toda una tabla de una base de datos, una lista de elementos XML o una colección de objetos .NET. En la clase Film la lista de actores es una colección de objetos de tipo string. La cantidad de controles necesarios para representar todos los elementos de una lista depende de la ejecución, por tanto no tiene sentido pretender establecer estáticamente los enlaces escribiendo código como Actor[1], Actor[2], etc. En WPF, el control para mostrar colecciones de tamaño variable es ItemsControl. La propiedad ItemsSource de éste permite enlazar al control con una fuente de datos que produzca una colección. Para usar este control en toda su potencialidad, hay que combinarlo con otras capacidades como las que se ven en la Lección **Plantillas de Datos**, por lo que aquí nos limitaremos a ver lo relacionado con los enlaces.

El Listado 6- 16 describe un Director con varios films.

Listado 6- 16 Declaración de un objeto Director con una lista de films

```

<ejemplo:Director      x:Key="Spielberg"      Nombre="Steven"      Spielberg"
Nacionalidad="USA">
  <ejemplo:Director.Filmes>
    <ejemplo:Film Titulo="Jurassik Park" Genero="Aventuras"
      Oscar="True" Calificación="4">
      <ejemplo:Film.Actores>
        <system:String>Sam Neill</system:String>
        <system:String>Laura Linney</system:String>
        <system:String>Jeff Goldblum</system:String>
        <system:String>Richard Athemboroug</system:String>
      </ejemplo:Film.Actores>
    </ejemplo:Film>
  </ejemplo:Director.Filmes>
</ejemplo:Director>

```

```

<ejemplo:Film Titulo="La Lista de Schindler" Genero="Drama"
    Oscar="True" Calificación="4.5">
    <ejemplo.Film Actores>
        <system:String>Lian Nesson</system:String>
        <system:String>Ben Kingsley</system:String>
        <system:String>Ralph Fiennes</system:String>
    </ejemplo.Film Actores>
</ejemplo:Film>
<ejemplo:Film Titulo="E.T." Genero="CienciaFicción" ...>...</ejemplo:Film>
<ejemplo:Film Titulo="Tiburón" ...>...</ejemplo:Film>
<ejemplo:Film Titulo="Color Púrpura" ...>...</ejemplo:Film>
<ejemplo:Film Titulo="Munich" ...>...</ejemplo:Film>
<ejemplo:Film Titulo="Encuentro Cercano de Tercer Tipo" ...>...</ejemplo:Film>
<ejemplo:Film Titulo="Inteligencia Artificial" ...>...</ejemplo:Film>
</ejemplo:Director Filmes>
</ejemplo:Director>

```

Para establecer el enlace con la lista de filmes se usa un control de tipo ItemsControl y la propiedad ItemsSource de éste como destino, tal como se muestra en el Listado 6- 17.

Listado 6- 17 Enlace con una colección de elementos para mostrar una lista

```

<StackPanel DataContext="{StaticResource Spielberg}" Margin="4">
    <TextBlock FontSize="14" FontWeight="Bold">Filmes</TextBlock>
    <ItemsControl ItemsSource="{Binding Filmes}" />
</StackPanel>

```

La visualización por defecto de la lista de filmes se muestra en la Figura 6- 11 a). Como el tipo de datos Film no es conocido por WPF, su representación como cadena es la que hace implícitamente .NET con el método ToString. Para mejorar esto habría que redefinir el método ToString en la clase Film (por ejemplo para que al menos devuelva el título haciendo return Titulo). En tal caso el resultado de desplegar el ItemsControl es el que se muestra en la Figura 6- 11 b).



Figura 6- 11 Visualización de la lista de filmes

Una visualización más atractiva se puede ver en la Lección **Plantillas de Datos**.

6.3.1 Comportamiento Maestro-Detalle

La visualización de datos en modo Maestro-Detalle es un patrón muy conocido en las aplicaciones de Bases de Datos. El patrón consiste en mostrar, por una parte, una colección de elementos (a modo de resumen), y por otra parte, una versión detallada de los datos del elemento que se seleccione de la colección (el resumen) que se está mostrando. La Figura 6- 12 nos muestra una versión simplificada de Maestro Detalle.

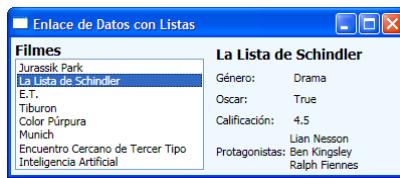


Figura 6- 12 Visualización Maestro-Detalle de la lista de filmes

Para lograr este patrón debemos visualizar el **Maestro** a través de algún control derivado de Selector, ya que el patrón exige que en la visualización de la colección de elementos, se pueda seleccionar uno cualquiera de los elementos, que será del que se visualizarán sus datos en la parte **Detalle**. En la Figura 6- 12 se ha utilizado un ListBox en lugar del ItemsControl del Listado 6- 17, como se ve en el Listado 6- 18.

Listado 6- 18 Enlace con una colección de elementos para mostrar una lista de selección de elementos

```
<StackPanel DataContext="{StaticResource Spielberg}" Margin="4">
    <TextBlock FontSize="14" FontWeight="Bold">Filmes</TextBlock>
    <ListBox Name="lbFilmes" ItemsSource="{Binding Filmes}" />
</StackPanel>
```

Hacer el enlace con un ListBox permite visualizar la lista de elementos que produce la fuente de datos y a su vez seleccionar uno de los elementos de la lista (el elemento seleccionado se publica en la propiedad SelectedItem del ListBox). Esta propiedad SelectedItem es la que servirá de fuente de datos a la parte **Detalle** del patrón para mostrar en más detalle los datos del elemento seleccionado.

El Listado 6- 19 nos define un enlace para el DataContext del Grid en el que la fuente de datos es el elemento seleccionado del elemento del ListBox de nombre lbFilmes (por simplicidad se han omitido los detalles del layout del Grid). Esto implica que cualquier cambio en la propiedad SelectedItem, producido en ejecución por una interacción con el ListBox, provoca un cambio inmediato en la propiedad DataContext del Grid, la cual a su vez se refleja en los enlaces dentro del Grid que hayan especificado un enlace con este DataContext. Una notación como {Binding Genero} indica que se tome como fuente el valor de DataContext y que de éste se tome el valor de la propiedad Genero.

Listado 6- 19 Uso de DataContext para implementar el patrón Maestro-Detalle

```
<Grid DataContext="{Binding SelectedItem, ElementName=lbFilmes}">
    ...
    <TextBlock Text="{Binding Titulo}" FontWeight="Bold" FontSize="14"
               Grid.ColumnSpan="2"/>
    <TextBlock Text="Género:" Grid.Row="1" Grid.Column="0"/>
    <TextBlock Text="{Binding Genero}" Grid.Row="1" Grid.Column="1"/>
    <TextBlock Text="Oscar:" Grid.Row="2" Grid.Column="0"/>
    <TextBlock Text="{Binding Oscar}" Grid.Row="2" Grid.Column="1"/>
    <TextBlock Text="Calificación:" Grid.Row="3" Grid.Column="0"/>
    <TextBlock Text="{Binding Calificación}" Grid.Row="3" Grid.Column="1"/>
    <TextBlock Text="Protagonistas:" Grid.Row="4" Grid.Column="0"/>
        <ItemsControl ItemsSource="{Binding Actores}" Grid.Row="4"
                      Grid.Column="1"/>
</Grid>
```

6.4 De cuando fuente y destino no concuerdan

Una razón por la que a veces no se puede crear directamente un enlace de datos es cuando los tipos de datos no son exactamente iguales o cuando no hay una conversión evidente (que pueda considerarse por WPF entre las predeterminadas). En este caso el uso de la interfaz IValueConverter permite a un enlace convertir los valores desde un extremo del enlace al otro (lo que podría hacerse para ambos sentidos). Otro caso en que puede ser útil la conversión de valores es cuando se desea convertir un dato para representarlo con una apariencia apropiada.

Aún cuando exista una conversión posible puede ser deseable que el valor a transportar por el enlace cumpla con determinada condición. Por ejemplo queremos comprobar que la calificación de un Film deba ser un valor entre 0 y 5. Para esto hay una solución en WPF que se basa en hacer validaciones de datos mediante la clase ValidationRule. De este modo se facilita que una violación de lo indicado en la validación se pueda notificar mediante algún artefacto visual.

6.4.1 Conversiones de Valores

La propiedad Converter de la clase Binding permite interceptar el proceso de transmisión de los datos a través de un enlace. Esta propiedad es de tipo IValueConverter, una interfaz definida en el espacio de nombres System.Windows.Data que declara dos métodos, Convert y ConvertBack, encargados de traducir los datos hacia uno y otro lado del enlace. En la Figura 6- 13 aparece un diagrama que ilustra cómo funciona el convertidor de valores en el enlace.

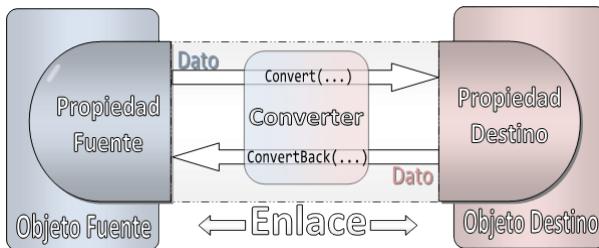


Figura 6- 13 Aplicación de los métodos Convert y ConvertBack en un enlace.

El Listado 6- 20 muestra la interfaz en C#.

Listado 6- 20 Interfaz IValueConverter

```
public interface IValueConverter {
    object Convert(object value, Type targetType, object parameter,
        CultureInfo culture);
    object ConvertBack(object value, Type targetType, object
        parameter,
        CultureInfo culture);
}
```

En muchos casos la aplicación de una conversión ocurre automáticamente, sin necesidad de declararlo explícitamente, este es el caso cuando a nivel de .NET existe una conversión elemental entre los dos tipos de datos de la fuente y el destino. En nuestro ejemplo de la clase Film, la propiedad Oscar es de tipo bool?, mientras que la propiedad Text del TextBlock o del TextBox es de tipo string. Evidentemente, la conversión de un tipo a otro ocurrió sin que esto se especificara explícitamente. Hay una variedad de clases que implementan esta interfaz IValueConverter, a las cuales no tenemos acceso públicamente, pero que se utilizan por WPF para realizar la conversión por defecto entre todos los tipos básicos.

En las librerías de WPF no existen muchas implementaciones públicas de conversores ya que por lo general no es necesario usarlas. No obstante para los más curiosos vamos a mostrar cómo implementar y utilizar nuestras propias conversiones.

En el ejemplo anterior vamos a cambiar al TextBlock que muestra la calificación de los filmes para que en lugar de poner un número muestre una secuencia de estrellas como se ilustra en la Figura 6- 14, donde la cantidad de estrellas rellenas en amarillo dependa del valor numéricico de la propiedad Calificacion.

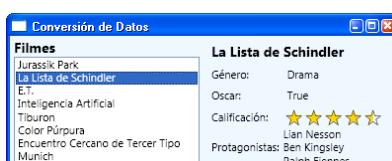


Figura 6- 14 Mostrar un dato mediante una conversión para visualizarlo

Después de estudiar las lecciones del Capítulo de **Gráficos** veremos cómo mejorar esta apariencia, por ahora vamos a conformarnos con la de la Figura 6- 14.

Para lograr el efecto de las estrellas utilizaremos tres rectángulos rellenos con diferentes patrones de figuras y colores, como muestra el diagrama de la Figura 6- 15. En el Capítulo **Gráficos** aprenderá como lograr las figuras y brochas que se han utilizado para dibujar estas estrellas.



Figura 6- 15 Al combinar en una casilla del Grid los tres rectángulos obtenemos tantas estrellas coloreadas según sea la calificación.

En el Listado 6- 21 se declara el recurso de nombre `rectWidthConverter` que se utiliza para convertir el dato de la calificación, que está en el rango de 0 a 5, a un valor que corresponda con el ancho que se le va a dar al rectángulo en amarillo que sirve para dar el efecto de relleno de las estrellas. Este recurso se utiliza desde la propiedad `Converter` del enlace resaltado en negritas. Cuando se hace

```
<Rectangle Width="{Binding Calificación,  
Converter={StaticResource rectWidthConverter}}"
```

se está indicando que el valor que se extraiga de la fuente `Calificación` se le aplique el conversor que se ha guardado en los recursos con la llave `rectWidthConverter`

Listado 6- 21 El ancho del rectángulo se enlaza a la calificación del film mediante un conversor.

```
<Window.Resources>
    <ejemplo:CalificacionToRangeConverter x:Key="rectWidthConverter"
        RangeLength="100"/>
    <DrawingBrush x:Key="starMask">...</DrawingBrush>
    <DrawingBrush x:Key="starBorder">...</DrawingBrush>
</Window.Resources>
...
...
<Grid DataContext="{Binding SelectedItem, ElementName=lbLibros}">
...
<TextBlock Text="Calificación:"/>
<Grid VerticalAlignment="Center" HorizontalAlignment="Left">
    <Rectangle Width="{Binding Calificación,
        Converter={StaticResource rectWidthConverter}}"
        Height="20" Fill="Gold" Margin="1" HorizontalAlignment="Left"/>
    <Rectangle Width="102" Height="22" Fill="{StaticResource starMask}"/>
    <Rectangle Width="102" Height="22" Fill="{StaticResource starBorder}"/>
</Grid>
</Grid>
...
```

La clase CalificaciónToRangeConverter se programa en el code-behind (Listado 6- 22). Esta clase no cambia el tipo del dato que convierte, ya que admite valores de tipo double y retorna valores del mismo tipo. Su función en este caso es convertir un valor de 0 a 5 que es la calificación de un film, a un valor de 0 a RangeLength (que en el ejemplo debe ser el ancho del rectángulo en el que se dibujan las estrellas). El segundo y tercer rectángulo se han trazado con ancho 102 para que cubran los bordes del rectángulo amarillo que yace debajo de ellos.

Listado 6- 22 Implementación de IValueConverter para convertir un valor de tipo double de un rango a otro

```
using System;
using System.Globalization;
using System.Text;
using System.Windows.Data;

namespace WPF_EnlaceDatos {
    public class CalificaciónToRangeConverter : IValueConverter {
        double rangeLength = 10;
        public double RangeLength { get {...} set {...} }
```

```

public object Convert(object value, Type targetType, object parameter,
    CultureInfo culture) {
    if (value == null) value = 0.0;
    double val = (double)value;
    if (targetType != typeof(double)) throw new
        ArgumentException("targetType");
    if (RangeLength == 0)
        throw new ArgumentException("FromLength can not be zero");
    return (val / 5.0) * RangeLength;
}
public object ConvertBack(object value, Type targetType, object
parameter,
    CultureInfo culture) {
    ...
}
}

```

El método Convert recibe cuatro parámetros que describimos a continuación:

1. object value: es el valor que se debe convertir. Aunque estáticamente el parámetro es de tipo object por lo general los conversores de tipo se implementan orientados a tipos más específicos dinámicamente. Esto hace que en ejecución, usando reflexión, se deba preguntar por el tipo que dinámicamente tendrá este parámetro value. En este ejemplo se asume que el tipo de datos de value debe ser double.
2. Type targetType: es el tipo de datos hacia el que hay que convertir el valor del parámetro value.

Los otros dos parámetros permiten pasarle al conversor información extra en el momento de convertir el valor.

El método ConvertBack tiene los mismos parámetros que Convert, pero se utiliza cuando el flujo del dato ocurre desde el destino hasta la fuente. El método ConvertBack solo tiene utilidad implementarlo si se va a utilizar en enlaces en modo OneWayToSource o TwoWay.

En el Listado 6- 23 se declara otro conversor que usaremos para representar el color del rectángulo de relleno en dependencia de la calificación del film (poner las estrellas en rojo cuando los films son de baja calificación y amarillo cuando la calificación es alta) y mostrarlos como se ilustra en la Figura 6- 16. En este caso, la fuente de datos genera valores en el rango de 0 a 5, que serán convertidos a valores a colores en un gradiente de colores desde un color inicial a un color final. En el Listado 6- 24 se muestra el conversor CalificaciónToColorConverter utilizado para lograr este efecto.



Figura 6- 16 La conversión de datos permite obtener elementos de apariencia como colores y brochas

Listado 6- 23 Uso de combinaciones de conversores

```

<Window.Resources
    xmlns:system="clr-namespace:System;assembly=mscorlib">
        <ejemplo:CalificaciónToRangeConverter
            x:Key="rectWidthConverter"
                RangeLength="100"/>
        <ejemplo:CalificaciónToColorConverter x:Key="colorConverter"
            StartColor="Red" EndColor="Gold"/>
        ...
</Window.Resources>
...
<Grid DataContext="{Binding SelectedItem, ElementName=lbLibros}">
    ...
        <TextBlock Text="Calificación:" Grid.Row="3" Grid.Column="0"/>
        <Grid VerticalAlignment="Center" HorizontalAlignment="Left">
            <Rectangle Width="{Binding Calificación,
                Converter={StaticResource rectWidthConverter}}"
                Fill="{Binding Calificación,
                Converter={StaticResource colorConverter}}"
                Height="20" Margin="1" HorizontalAlignment="Left"
            />
            <Rectangle Width="102" Height="22" Fill="{StaticResource
                starMask}"/>
            <Rectangle Width="102" Height="22" Fill="{StaticResource
                starStroke}"/>
        </Grid>
    </Grid>

```

Listado 6- 24 Esta clase permite convertir un valor double en el rango de 0 a 1 en una brocha sólida de un color entre StartColor y EndColor

```
public class CalificaciónToColorConverter : IValueConverter {
    Color startColor = Colors.White;
    Color endColor = Colors.Blue;
    public Color StartColor { get { ... } set { ... } }
    public Color EndColor { get { ... } set { ... } }
    public object Convert(object value, Type targetType, object parameter,
        CultureInfo culture) {
        double val = value is double ? (double)value : 0.0;
        if (val < 0) val = 0;
        if (val > 5) val = 5;
        val /= 5.0;
        Color color = Color.FromArgb(Mid(startColor.A, endColor.A, val),
            Mid(startColor.R, endColor.R, val),
            Mid(startColor.G, endColor.G, val),
            Mid(startColor.B, endColor.B, val));
        return new SolidColorBrush(color);
    }
    byte Mid(byte start, byte end, double ratio) {
        double val = start + (end - start) * ratio;
        if (val < 0) val = 0;
        if (val > 255) val = 255;
        return (byte)val;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
        CultureInfo culture) {
        ...
    }
}
```

6.4.2 Validación de la entrada de datos

Mediante los conversores de tipos, podemos implementar también una lógica de validación. Veamos cómo garantizar que a la calificación sólo se le puedan dar un valor entre 0 y 5.

Para tales situaciones se dispone de la clase ValidationRule, definida en el espacio de nombres System.Windows.Controls. Con esta clase se puede verificar una entrada de datos en el destino

de datos (por ejemplo el control de interacción), antes incluso de intentar convertir el dato camino a la fuente (por ejemplo el objeto de negocio).

No se confunda, seguimos diciendo destino y fuente en términos del **destino** como elemento de interfaz WPF y la **fuente** como el objeto de negocios .NET, independientemente de si el elemento de interfaz lo usamos para introducir un dato que termine provocando una actualización en el objeto de negocio como es el caso en este ejemplo de validación.

La Figura 6- 17 muestra la situación en la que se utilizan las reglas de validación.

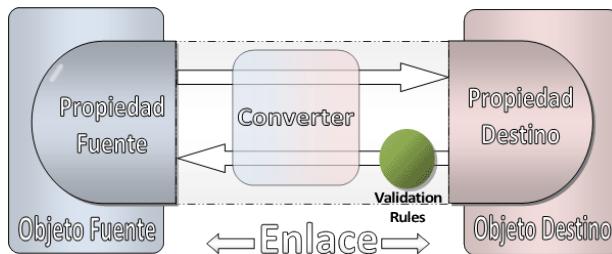


Figura 6- 17 Uso de reglas de validación en un enlace.

La clase Binding tiene también la propiedad ValidationRules, que es una colección de objetos de tipo ValidationRule. Un ValidationRule es una clase abstracta de la que se debe implementar su único método Validate.

WPF ofrece un heredero de ValidationRule que es la clase ExceptionValidationRule. Esta clase permite tratar cualquier excepción que se produzca al intentar transportar el valor desde el destino hasta la fuente, no importa en que parte de este camino se produzca. El código XAML del Listado 6- 25 muestra la declaración de un enlace con una regla de validación para el tratamiento de excepciones.

Listado 6- 25 Uso de ExceptionValidationRule para capturar excepciones desde las fuentes de datos y visualizarlas en la interfaz de usuario

```
<TextBlock Text="Género:" Grid.Row="1" Grid.Column="0"/>
<TextBox Grid.Row="1" Grid.Column="1">
    <TextBox.Text>
        <Binding Path="Genero">
            <Binding.ValidationRules>
                <ExceptionValidationRule/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

El Listado 6- 25 indica que se aplique un ExceptionValidationRule observe en la Figura 6- 18 cómo después de escribir un valor inválido en el género del film, se muestra un rectángulo rojo

alrededor del control destino del enlace que viola la regla. Aquí WPF ha aplicado el conversor predeterminado que convierte de string a enumerativo (Observe que según el Listado 6-3 el género se ha definido como del tipo enumerativo Genero y este no incluye un tal valor DramaBélico). Al violarse una regla, las propiedades adjuntas HasError y Errors definidas en la clase Validation toman valores acordes a la situación. Es posible mediante el uso de plantillas de controles (Lección **Plantillas de Controles**) cambiar la manera en que se visualiza el error.



Figura 6- 18 Antes y después de escribir un valor inválido para una propiedad que no admite el valor

Cuando los objetos de negocio no implementan una validación o cuando en la interfaz de usuario se desea imponer una validación adicional, es posible colocar en el XAML nuestras propias validaciones. Por ejemplo, queremos garantizar que los filmes se tengan que clasificar con un valor entre 0 y 5. Para ello vamos a usar la regla RangeValidationRule definida en el Listado 6- 27 que hereda de la clase ValidationRule (Listado 6- 26) y añadimos esta validación a la lista de validaciones de la propiedad Calificación (Listado 6- 28).

Listado 6- 26 Definición de la clase ValidationRule

```
public abstract class ValidationRule {
    protected ValidationRule() {}
    public abstract ValidationResult Validate(object value, CultureInfo
cultureInfo);
}
```

El parámetro value recibe el valor que se debe validar y la respuesta del método debe ser una instancia de ValidationResult que tiene un par de propiedades: bool IsValid y object ErrorContent. Dentro de este método no se debe lanzar excepción.

Listado 6- 27 Definición de una regla de validación

```
public class RangeValidationRule : ValidationRule {
    double? minValue;
    double? maxValue;
    public double? MinValue {
        get { return minValue; }
        set { minValue = value; }
    }
}
```

```

public double? MaxValue {
    get { return maxValue; }
    set { maxValue = value; }
}
public override ValidationResult Validate(
    object value,
    System.Globalization.CultureInfo cultureInfo) {
    string str = value as string;
    if (str != null) {
        int val;
        if (int.TryParse(str, out val)) {
            if (minValue.HasValue && minValue > val) return new
ValidationResult(false,
                "Value must be a value greather than or equals to " + minValue);
            if (maxValue.HasValue && maxValue < val) return new
ValidationResult(false,
                "Value must be a value less than or equals to " + maxValue);
            return new ValidationResult(true, null);
        }
    }
    return new ValidationResult(false,
        "Value must be a string representing a number");
}
}

```

Listado 6- 28 Uso de una regla de validación definida por el usuario

```

<TextBlock Text="Calificación:" Grid.Row="3" Grid.Column="0"/>
<TextBox Grid.Row="3" Grid.Column="1">
    <TextBox.Text>
        <Binding Path="Calificación">
            <Binding.ValidationRules>
                <ExceptionValidationRule/>
                <ejemplo:DoubleRangeValidationRule MinValue="0" MaxValue="5"/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>

```

Capítulo III GRÁFICOS

Este capítulo comienza con las figuras y los recursos básicos para hacer figuras. Luego se tratan las brochas y pinceles con los cuales podemos "pintar" nuestros elementos. También se tratan los efectos que se pueden aplicar sobre los elementos visuales y las transformaciones gráficas a las cuales estos pueden someterse. WPF basa la visualización de todos sus elementos en la tecnología gráfica Microsoft DirectX de modo que los amantes de las aplicaciones gráficas tienen ahora facilidades para tocar el paraíso con sus dedos.

Lección 7 Figuras

Poder dotar a las aplicaciones de una apariencia atractiva es uno de los objetivos fundamentales de WPF. Las figuras es uno de los principales aportes de WPF a las interfaces de usuario, y la facilidad con que se pueden definir figuras es una de las grandes atracciones de WPF para los desarrolladores de software.

En toda interfaz de usuario las figuras nos dan el primer impacto positivo de la aplicación. La combinación coherente, simple y atractiva de los elementos gráficos aporta estímulo, comprensibilidad y comodidad en el uso a las aplicaciones.

Las figuras son los elementos geométricos que de hecho componen la gran mayoría de los controles y demás elementos visuales. Son el punto de partida más básico si queremos definir nuestros propios controles y elementos visuales desde cero. En esta lección daremos una panorámica de las principales figuras de WPF.

7.1 Figuras básicas

Imagine que Ud. se encuentra frente a una hoja en blanco con un lápiz en la mano y quiere dibujar figuras geométricas. Con buen pulso podrá lograr un esbozo aproximado de figuras como el rectángulo, un triángulo, una línea recta o una circunferencia. Sin embargo Ud. coincidirá con nosotros que es más práctico y seguro emplear instrumentos de trazado como cartabones, reglas, compases o plantillas de figuras. Justamente esto son las figuras de WPF: instrumentos de trazado y plantillas de figuras de alta precisión. En la Figura 7-1 se muestran una línea con efectos de trazado y relleno, un rectángulo, una elipse y un polígono con efecto de sombreado. En esta lección conoceremos como lograr estas y otras figuras.

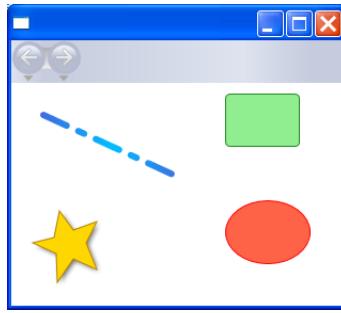


Figura 7-1 Ejemplos de figuras de WPF

Las figuras en WPF están en la misma categoría que los controles y los paneles y con las mismas posibilidades que el resto de los elementos de interfaz de usuario: Las figuras son elementos de tipos herederos de Shape que tiene a su vez como tipo base FrameworkElement. De este modo si Ud. quiere poner por ejemplo un rectángulo en la parte izquierda de su aplicación sólo necesita ubicarlo en un Dockpanel como con a cualquier otro elemento. En la siguiente sección veremos en más detalle un rectángulo esta figura que está presente en muchos elementos de interfaz.

7.1.1 Rectángulo

El rectángulo es una de las figuras más empleadas en la apariencia de las aplicaciones. El tipo Rectangle es el que representa a los rectángulos. Como las demás figuras los rectángulos tienen una propiedad Fill de tipo Brush para indicar cómo debe rellenarse y una propiedad Stroke, también de tipo Brush, para indicar cómo debe dibujarse su contorno. Además los rectángulos tienen Width y Height que indican su ancho y su altura. El siguiente XAML muestra cómo poner un rectángulo en un panel Canvas.

Listado 7-1 Rectángulo definido en XAML
<pre><Page x:Class="DrawingSample.Shapes2" Title="Shapes2"> <Canvas> <Rectangle Canvas.Left="20" Canvas.Top="20" Width="100" Height="60" Stroke="Green" Fill="LightGreen"/> </Canvas> </Page></pre>

La Figura 7-2 muestra el rectángulo color verde claro con de tamaño 100 x 60 a 20 puntos de la esquina izquierda y superior del área del panel.

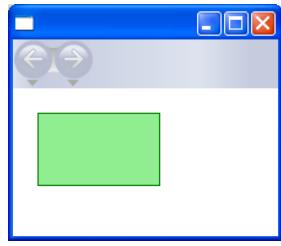


Figura 7-2 Rectángulo en un Canvas.

Al igual que otros elementos, las figuras pueden ser ubicadas en cualquier panel. Ubiquemos ahora (Listado 7-2) el rectángulo alineado a la izquierda en un DockPanel (Figura 7-3). Note que ahora no hace falta definir el alto (Height). De hacerlo el rectángulo no tomaría todo el alto de la ventana.

Listado 7-2 Rectángulo que se redimensiona acorde con el panel donde está ubicado

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
  <DockPanel HorizontalAlignment="Left">
    <Rectangle DockPanel.Dock="Left" Width="100"
      Stroke="Green" Fill="LightGreen"/>
  </DockPanel>
</Page>
```

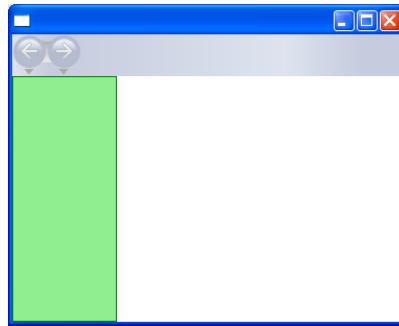


Figura 7-3 Rectángulo alineado a la izquierda en un DockPanel

En las interfaces gráficas de usuario de la mayoría de las aplicaciones modernas se emplean mucho los rectángulos con esquinas redondeadas. El Listado 7-3 muestra cómo hacer esto mediante las propiedades RadiusX y RadiusY de Rectangle para lograr un rectángulo como el de la Figura 7-4:

Listado 7-3 Rectángulo con esquinas redondeadas

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
  <Canvas>
    <Rectangle Canvas.Left="20" Canvas.Top="20" Width="100"
      Height="60"
      Stroke="Green" Fill="LightGreen"
      RadiusX="10" RadiusY="10"/>
  </Canvas>
</Page>
```

```

    RadiusX="10" RadiusY="10"/>
</Canvas>
</Page>

```

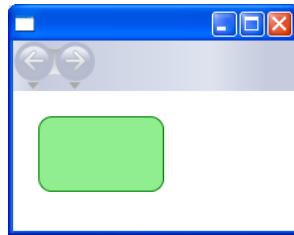


Figura 7-4 Rectángulo con esquinas redondeadas

Las propiedades `RadiusX` y `RadiusY` refieren a los dos radios que se aplican a una elipse. La Figura 7-5 muestra el significado de cada una de estas propiedades.

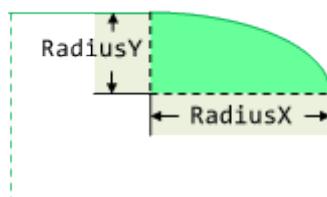


Figura 7-5 Significado de las propiedades `RadiusX` y `RadiusY` de `Rectangle`

Si se aplican valores distintos a los radios, por ejemplo en `RadiusX="30"` y en `RadiusY="10"` el resultado es como el que se puede ver en la Figura 7-6.

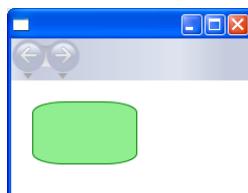


Figura 7-6 Rectángulo con `RadiusX` y `RadiusY` distintos

7.1.2 Elipse

El tipo `Ellipse` representa una elipse que se inscribe en un rectángulo imaginario de dimensiones `Width` x `Height` (Figura 7-7).

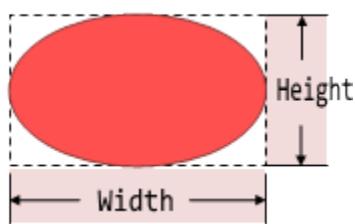


Figura 7-7 Elipse definida a partir de Width y Height

Las elipses se declaran de la misma forma que los rectángulos. La Figura 7-8 muestra una elipse color tomate con borde marrón de tamaño 100 x 60 a 20 puntos de los bordes izquierdo y superior.

Listado 7-4 Elipse definida en XAML

```
<Page x:Class="DrawingSample.Shapes2"
      Title="Shapes2">
    <Canvas>
      <Ellipse Width="100" Height="60"
               Stroke="Maroon" Fill="Tomato"/>
    </Canvas>
  </Page>
```

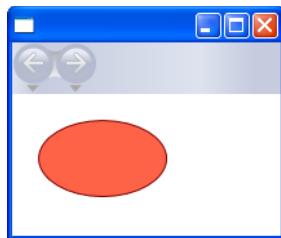


Figura 7-8 Elipse

7.1.3 Línea

Las líneas son menos usadas que las elipses y los rectángulos porque sus dimensiones no son ajustables según la distribución de los paneles: Tanto en el rectángulo como en la elipse las dimensiones horizontal (Width) como la vertical (Height) se pueden ajustar según su panel contenedor en correspondencia con las propiedades VerticalAlignment y HorizontalAlignment, sin embargo la línea no tiene Width o Height sino solo las coordenadas X1, Y1, X2 y Y2, las cuales indican el punto de origen (X1,Y1) y el punto final (X2,Y2). Aunque las líneas tienen la propiedad Fill porque heredan de Shape claro está que esta propiedad no tiene utilidad en Line, las líneas sólo tienen contorno se muestran siempre con la brocha que se indica en la propiedad Stroke. El siguiente XAML muestra la línea de la Figura 7-9.

Listado 7-5 Línea definida en XAML

```
<Page x:Class="DrawingSample.Shapes2"
      Title="Shapes2">
    <Canvas>
      <Line X1="20"   Y1="20"   X2="100"   Y2="80"
            Stroke="Blue"/>
    </Canvas>
  </Page>
```

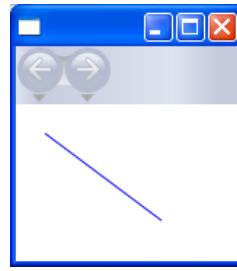


Figura 7-9 Línea azul

Pero Ud. no se decepcione si quiere lograr el efecto de una línea que se ajuste al tamaño de un panel. Es posible lograrlo pero no con una línea, sino con un rectángulo. En la sección **Decoradores** mostramos una manera de ajustar elementos como la línea al tamaño del panel contenedor. Si por ejemplo Ud. quiere dibujar un separador vertical entre botones, como el que suele usarse en un ToolBar para separar grupos de botones, puede lograrlo de la forma que se muestra en el Listado 7-6. Este nos despliega la Figura 7-10 que nos muestra un Grid donde la primera fila se ajusta automáticamente al tamaño de sus componentes. Note que hay dos rectángulos, ninguno de ellos tiene especificado su alto, por lo cual el tamaño de la fila se ajusta al tamaño y margen de los botones. La línea separadora se logra entonces con un rectángulo de ancho con ancho entre 1 y 2.

Listado 7-6 Rectángulo utilizado como separador simulando una línea

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>
</Grid.RowDefinitions>
<Rectangle RadiusX="4" RadiusY="4" Fill="LightGray" Stroke="Gray"/>
<StackPanel Orientation="Horizontal">
<Button FontWeight="Bold" Margin="2,5" Width="30">B</Button>
<Button FontStyle="Italic" Margin="2,5" Width="30">I</Button>
<Rectangle Width="2" Fill="DarkGray" Margin="5"/>
<Button Margin="2,5" Padding="5,2">Close</Button>
</StackPanel>
</Grid>
</Page>
```



Figura 7-10 Línea separadora dibujada usando un rectángulo

7.1.4 Polígonos y poligonales

Además de las figuras elementales como el rectángulo, la elipse y la línea, se cuenta en WPF con otras figuras que nos ayudan a enriquecer las interfaces de usuario. Usted seguramente ha visto las famosas estrellitas de elementos preferidos en aplicaciones como el Media Player:



¿Qué tal si queremos lograr tales estrellitas en nuestra aplicación?

Los polígonos y poligonales permiten dibujar figuras formadas por la unión lineal de un conjunto de puntos. La diferencia entre un polígono (`Polygon`) y una poligonal (`Polyline`) es que el primero es cerrado, o sea que el primero y último puntos se unen también, mientras que las poligonales no se unen, son abiertas.

En el siguiente ejemplo (Listado 7-7) que nos produce la Figura 7-11 se muestra como se define una estrella en XAML.

Listado 7-7 Polígono que describe una estrella

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
  <Canvas>
    <Polygon Points="10,40 32,40 40,10 48,40 70,40 53,53 60,80 40,65 20,80
27,53"
      Fill="Gold" Stroke="DarkGoldenrod"/>
  </Canvas>
</Page>
```

En .NET el tipo correspondiente `Polygon` tiene una propiedad `Points` de tipo array `Point[]` al que se le puede dar valor en XAML mediante una secuencia de puntos X,Y separados por espacio. WPF convierte este string que denota una secuencia de pares X,Y en un array de `Point` que es lo que se le asigna a esta propiedad `Points`. .

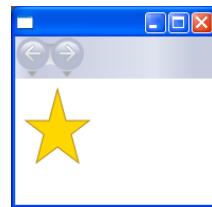


Figura 7-11 Estrella lograda con un polígono.

Ahora podríamos lograr la figura con las 5 estrellitas si usamos un StackPanel horizontal en el que copiemos 5 veces un código como el del polígono del Listado 7-7.

Producto de la definición explícita de las coordenadas que forman a los polígonos y poligonales, al igual que con las líneas, estos elementos no se ajustan a las dimensiones del elemento que los contiene. En la sección **Decoradores** se estudia el elemento ViewBox que precisamente tiene como función hacer ajuste de dimensiones.

7.2 Decoradores

Los decoradores son elementos que juegan un papel muy parecido al de las figuras y a su vez funcionan de modo parecido a los paneles. Un decorador es un elemento de interfaz de usuario que, al igual que los paneles, puede contener elementos, pero a diferencia de estos los decoradores solo pueden contener un elemento. Cada decorador aplica una forma de decorar al elemento que contiene. El más usado de los decoradores es el Border.

7.2.1 Border

El Border es un elemento que enmarca en un rectángulo al elemento que contiene aunque no tiene como tal un elemento rectángulo visualmente Ud. puede imaginar el decorador como un rectángulo con contenido aunque el decorador tiene propiedades que no tienen los rectángulos y viceversa. El siguiente XAML define un Border que enmarca a un Label (Figura 7-12)

Listado 7-8 Border con esquinas redondeadas y un bloque de texto en su interior

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
  <Canvas>
    <Border Width="100" Height="40" CornerRadius="10" Margin="20"
           BorderBrush="RoyalBlue"             Background="AliceBlue"
           BorderThickness="1">
      <Label VerticalAlignment="Center" HorizontalAlignment="Center">
        Texto centrado
      </Label>
    </Border>
  </Canvas>
</Page>
```

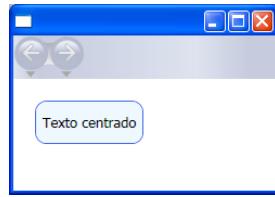


Figura 7-12 Borde que contiene un Label

Note que al Label contenido en el Border se le han indicado alineaciones horizontal y vertical. Fíjese como el borde hace que su contenido se ubique en su interior acorde con estas propiedades como mismo lo hacen los paneles.

En el Border, al igual que en los controles, la propiedad para indicar el relleno se llama Background y no Fill como en las figuras, y la propiedad que permite indicar el color del contorno es BorderBrush en vez de Stroke.

Los bordes además tienen la propiedad CornerRadius que tiene un efecto parecido a RadiusX y RadiusY del rectángulo: Con CornerRadius Ud. puede indicar el grado de redondez de las esquinas del borde. A diferencia del rectángulo Ud. no puede definir radios distintos para la X y la Y en cada esquina, cada esquina se muestra como un arco de circunferencia y no de elipse. Sin embargo, CornerRadius permite indicar un radio distinto en cada esquina. El Listado 7-9 ilustra como usando la propiedad CornerRadius se puede simular los encabezamientos de un TabControl (Figura 7-13).

Listado 7-9 Borde con sólo dos esquinas redondeadas que simula un encabezamiento de un TabControl

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
  <Canvas>
    <Border Width="100" Height="40" CornerRadius="10,10,0,0" Margin="20"
           BorderBrush="RoyalBlue"             Background="AliceBlue"
           BorderThickness="1">
      <Label VerticalAlignment="Center" HorizontalAlignment="Center">
        Texto centrado
      </Label>
    </Border>
  </Canvas>
</Page>
```

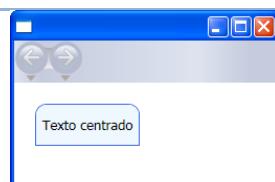


Figura 7-13 Borde con dos esquinas redondeadas

Note que CornerRadius ha sido definido aquí a partir de la cadena "10,10,0,0". El primer número indica el grado de redondez de la esquina superior izquierda, siguiendo el sentido de las manecillas del reloj. El siguiente número es la redondez del borde superior derecho, luego el inferior derecho y finalmente el inferior izquierdo. Al indicar 10, 10, 0 y 0 se ha dicho que las esquinas superiores tienen redondez 10 y las de abajo no tienen redondez.

7.2.2 Viewbox

El ViewBox es un decorador que Ud. puede emplear cuando necesite ajustar imágenes o figuras al tamaño del elemento que las contenga. A diferencia del Border la "decoración" del ViewBox no añade elementos gráficos al elemento decorado. Su función es la de ajustar el tamaño del elemento contenido a las dimensiones del ViewBox tomando en cuenta diferentes criterios que se indican a través de su propiedad Stretch.

Pongamos por caso que se quiera poner una imagen que se ajuste a determinadas dimensiones de un elemento como se ilustra en la Figura 7-14. Para esto Ud. puede simplemente usar un ViewBox como en el Listado 7-10.

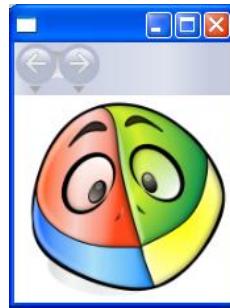


Figura 7-14 Imagen ajustada a un ViewBox

Listado 7-10 Imagen decorada con un ViewBox

```
<Page x:Class="DrawingSample.Shapes8"

      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Shapes8" >
    <Viewbox>
      <Image Source="Images/Windito.png"/>
    </Viewbox>
</Page>
```

Este ajuste de la imagen a las dimensiones de la ventana ocurre gracias a la combinación de las propiedades de alineación (HorizontalAlignment y VerticalAlignment) y a la propiedad de ajuste (Stretch) del ViewBox. Aunque no hemos puesto ninguna de estas propiedades en el Listado 7-10, por defecto HorizontalAlignment y VerticalAlignment tienen valor "Stretch" por lo cual el

ViewBox toma las dimensiones de la ventana. A su vez, el valor por defecto de la propiedad Stretch del ViewBox es "Uniform", lo cual indica que el elemento contenido dentro del ViewBox debe ajustarse dentro del ViewBox de forma que cubra la mayor parte de las dimensiones horizontal o vertical del ViewBox de modo que el elemento aparezca contenido totalmente en su interior pero sin perder sus proporciones. Los demás valores que puede tomar esta propiedad puede observarlos en la Figura 7-15 y Listado 7-11. Se muestran diferentes ViewBox cada uno en un Border que se ajusta a las dimensiones de las celdas de un Grid. Observe que en este caso es el Border el que decora gráficamente los ViewBoxes para que la imagen aparezca dentro de un rectángulo contorneado de negro como en la Figura 7-15.

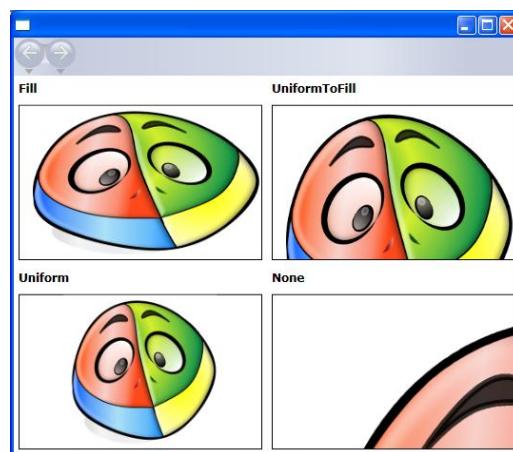


Figura 7-15 Diferentes formas de ajustar un elemento en un Viewbox usando los valores de Stretch

Listado 7-11 Cuatro ViewBoxes mostrándose con diferente valores de la propiedad Stretch

```
<Page x:Class="DrawingSample.Shapes8"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Shapes8" >
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>
<RowDefinition Height="Auto"/>
<RowDefinition/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition/>
<ColumnDefinition/>
</Grid.ColumnDefinitions>
```

```

<TextBlock FontWeight="Bold" Margin="5" >Fill</TextBlock>
<Border BorderBrush="Black" Margin="5" BorderThickness="1"
Grid.Row="1">
<Viewbox Stretch="Fill">
<Image Source="Images/Windito.png"/>
</Viewbox>
</Border>
<TextBlock FontWeight="Bold" Grid.Column="1" Margin="5">
UniformToFill
</TextBlock>
<Border BorderBrush="Black" Margin="5" BorderThickness="1"
Grid.Column="1"
Grid.Row="1">
<Viewbox Stretch="UniformToFill">
<Image Source="Images/Windito.png"/>
</Viewbox>
</Border>
<TextBlock FontWeight="Bold" Grid.Row="2"
Margin="5">Uniform</TextBlock>
<Border BorderBrush="Black" Margin="5" BorderThickness="1"
Grid.Row="3">
<Viewbox Stretch="Uniform" >
<Image Source="Images/Windito.png"/>
</Viewbox>
</Border>
<TextBlock FontWeight="Bold" Grid.Column="1" Grid.Row="2"
Margin="5">
None
</TextBlock>
<Border BorderBrush="Black" Margin="5" BorderThickness="1"
Grid.Column="1"
Grid.Column="1"
Grid.Row="3">
<Viewbox Stretch="None">
<Image Source="Images/Windito.png"/>
</Viewbox>
</Border>
</Grid>
</Page>

```

A continuación mostramos una breve descripción del efecto que provoca cada uno de los valores que puede tomar la propiedad Stretch:

Fill: Indica que la imagen debe cubrir toda el área del ViewBox, ajustándose completamente a sus dimensiones horizontal y vertical, esto hace que la imagen pueda perder sus proporciones originales como se aprecia en la figura.

None: La imagen no se extiende en lo absoluto y se muestra con su tamaño original. La Figura 7-15 muestra como se rellenaría el fondo de la página con Stretch="None".

UniformToFill: El valor UniformToFill en la propiedad Stretch hace que la imagen se ajuste a la mayor de las dimensiones (vertical u horizontal) para ajustarse al tamaño del ViewBox sin perder sus proporciones. Note en la Figura 7-15 que la imagen se redimensiona cubriendo el fondo del rectángulo, pero sin mostrarse completamente. Para que la imagen quedara proporcional y completa dentro del ViewBox las dimensiones de éste tendrían que ser proporcionales a las dimensiones del elemento que se está decorando.

Uniform: Este es el valor por defecto de Stretch en el ViewBox. Con Uniform la imagen se ajusta a la menor de las dimensiones vertical u horizontal del ViewBox de modo que la imagen siempre aparece completamente contenida dentro del área de relleno sin perder sus proporciones. La diferencia entre Uniform y UniformToFill se hace aún más evidente cuando las proporciones del ViewBox no coinciden con las del elemento que este contiene. Con ambos valores se mantienen las proporciones del elemento pero como con Uniform debe garantizarse que el elemento se vea completamente en ocasiones queda una porción del área de relleno sin cubrir. Observe en la Figura 7-15 cómo al llenar el rectángulo con extensión Uniform han quedado dos espacios sin llenar a ambos lados de la imagen.

Usted puede utilizar el ViewBox en todos los casos en que necesite ajustar figuras que se definen con coordenadas fijas. La línea vertical que originalmente queríamos poner como separador en la botonera ahora poniéndola dentro de un ViewBox podemos lograr el efecto de ajustarla al tamaño (vertical) de su panel contenedor sin tener que poner un valor fijo en las coordenadas de la línea. También podría utilizar el ViewBox para redimensionar fácilmente el polígono de la Figura 7-11 a su gusto.

7.3 Figuras vs Controles

En algunos casos es fácil lograr rápidamente un dibujo que tenemos en mente a partir de configurar controles. Si por ejemplo quisiéramos poner un texto enmarcado en un rectángulo con fondo verde claro y contorno verde oscuro, podemos lograrlo con facilidad modificando directamente propiedades del Label (Listado 7-10, Figura 7-16).

Listado 7-12 Label que muestra un texto con fondo verde

```
<Page x:Class="DrawingSample.Shapes2" Title="Shapes2">
<Canvas>
  <Label Canvas.Left="20" Canvas.Top="20" Width="160" Height="30"
```

```

        Background="LightGreen"           BorderBrush="DarkGreen"
BorderThickness="1">
    Texto bordeado de verde.
</Label>
</Canvas>
</Page>

```

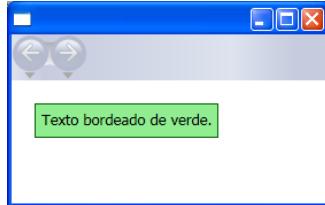


Figura 7-16 Label con propiedades de apariencia modificadas

En la Lección **Plantillas de Controles** se muestra cómo los controles pueden cambiar su aspecto dinámicamente al cambiar su estilo. Una etiqueta de fondo verde claro y contorno verde oscuro no indica que la figura que lo enmarca tenga que ser un rectángulo, sino sólo que la figura que contenga al texto debe tener estos colores para su fondo y su contorno. De este modo cambiando el estilo del Label se puede lograr que en vez de un rectángulo, el texto esté enmarcado en una elipse o un contorno redondeado, o incluso una imagen con saturación verde.

Ahora bien, el objetivo fundamental de los controles no es exactamente hacer de elementos gráficos sino que son elementos de interacción con el usuario, es decir que reaccionan a acciones del usuario y producen eventos. De modo que si lo que Ud. realmente quiere es solamente que se muestre un rectángulo verde con un texto dentro, le recomendamos que en lugar de un Label use un Border que contenga un TextBlock.

7.4 Figuras Avanzadas

En sus aplicaciones Ud puede querer no sólo figuras básicas como rectángulos, elipses o líneas. WPF permite dibujar casi cualquier figura a través del tipo Path. El tipo Path define una figura que se obtiene por composición de múltiples elementos geométricos como rectángulos, elipses, líneas, arcos, beziers o a su vez figuras combinadas.

Iremos de menos a más describiendo algunas de las bondades de este tipo de figuras.

Para los que conozcan GDI+: El tipo Path es parecido a GraphicsPath

7.4.1 Figuras combinadas

Para ilustrar las figuras combinadas imagínese que se quiera dibujar una figura como la de un número 8 (Figura 7-17). Esta figura puede verse como la unión de todos los puntos de dos circunferencias.

Path es un elemento mediante el cual se puede dibujar una figura como combinación de otras. En este caso la figura que se muestra en la Figura 7-15 se obtiene mediante el Path del Listado 7-11, éste contiene un CombinedGeometry que a su vez contiene dos elipses (circunferencias). Cuando el CombinedGeometry no indica explícitamente como combinar las figuras que contiene WPF considera por defecto una figura que es la unión de ambas figuras contenidas (en este ejemplo las dos elipses).

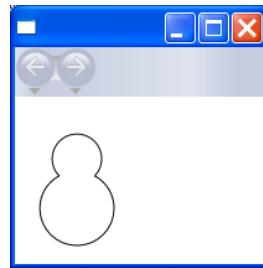


Figura 7-17 Silueta del número 8 formada por combinación de elipses.

Listado 7-13 XAML que describe el dibujo de la Figura 7-17

```
<Page x:Class="DrawingSample.Shapes5"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Shapes5" >
  <Grid>
    <Path Stroke="Black">
      <Path.Data>
        <CombinedGeometry>
          <CombinedGeometry.Geometry1>
            <EllipseGeometry Center="50,50" RadiusX="20" RadiusY="20"/>
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
            <EllipseGeometry Center="50,90" RadiusX="30" RadiusY="30"/>
          </CombinedGeometry.Geometry2>
        </CombinedGeometry>
      </Path.Data>
    </Path>
  </Grid>
</Page>
```

Un CombinedGeometry puede indicar explícitamente en la propiedad GeometryCombineMode la forma en que combina las figuras. La Figura 7-16 nos muestra las distintas posibilidades de cómo combinar las dos elipses del Listado 7-13. Hemos rellenado el fondo de la figura de color LightSteelBlue para que usted pueda notar más claramente el efecto de cada valor en la combinación.

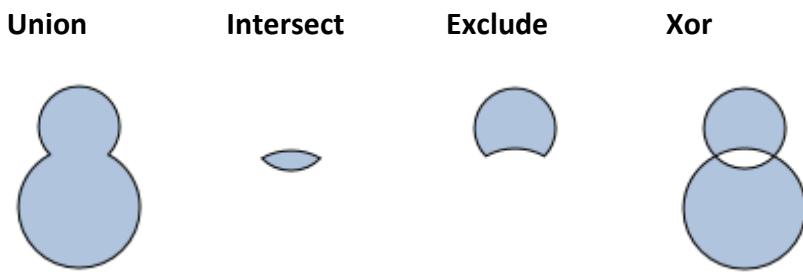


Figura 7-18 Diferentes valores de la propiedad GeometryCombineMode

Path puede contener diferentes figuras geométricas en su propiedad Data. Las figuras geométricas son una forma precisa de definir figuras pero no son elementos de interfaz de usuario (no son UIElements), sino que son primitivas que se emplean para definir los contornos de las figuras. Así por ejemplo RectangleGeometry es la figura geométrica que usa Rectangle para definir su contorno, como Ellipse usa EllipseGeometry y Line usa a LineGeometry. Además hay figuras como CombinedGeometry que permite formar una figura a partir de la combinación de otras dos figuras asignándolas a sus propiedades Geometry1 y Geometry2.

Ud puede formar prácticamente cualquier figura combinándolas dos a dos. por ejemplo para combinar tres figuras, usted puede formar una figura combinada por composición de una primera figura simple y otra figura combinada compuesta por otras dos figuras simples.

7.4.2 PathGeometry

El tipo PathGeometry define la figura geométrica primitiva de la figura Path y es a través de esta geometría que se alcanza la mayor expresividad visual de las figuras 2D en WPF.

Un PathGeometry define una figura plana formada a partir de la unión de segmentos (líneas, arcos, cuadráticos y curvas de bezier). Una estrella como la que definimos con un Polygon puede formarse también a partir de la unión de diez segmentos de línea.

Los 4 tipos de segmentos básicos que se pueden formar se ilustran en la Figura 7-19 resultado de ejecutar el código del Listado7-14. Para indicar los segmentos que forman un PathGeometry se declara en su interior PathFigure que es la colección de segmentos del PathGeometry. La utilidad principal del PathFigure es su propiedad StartPoint donde debe indicarse el punto inicial a partir del cual se construirá la figura. El resto de la figura se construye añadiendo segmentos de figura en el interior del PathFigure. El punto inicial de cada segmento será el punto final del

segmento anterior y el primer segmento comienza en el punto que se haya indicado en la propiedad StartPoint del PathFigure.

Listado 7-14 Ejemplo de Path simples que muestran cada uno un tipo diferente de segmento de figura.

```
<Page x:Class="DrawingSample.Shapes6"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Shapes6" >
<Grid>
<Grid.ColumnDefinitions>
    <ColumnDefinition/> <ColumnDefinition/>
<ColumnDefinition/>
    <ColumnDefinition/>
</Grid.ColumnDefinitions>
<Path Stroke="Black">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="5,5">
                <LineSegment Point="50,50"/>
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
<Path Stroke="Black" Grid.Column="1">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="5,5">
                <ArcSegment Point="50,50" Size="45,25"/>
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
<Path Stroke="Black" Grid.Column="2">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="5,5">
                <QuadraticBezierSegment Point1="50,15" Point2="50,50" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
```

```

<Path Stroke="Black" Grid.Column="3">
  <Path.Data>
    <PathGeometry>
      <PathFigure StartPoint="5,5">
        <BezierSegment Point1="50,15" Point2="15,50"
          Point3="50,50"/>
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
</Grid>
</Page>

```

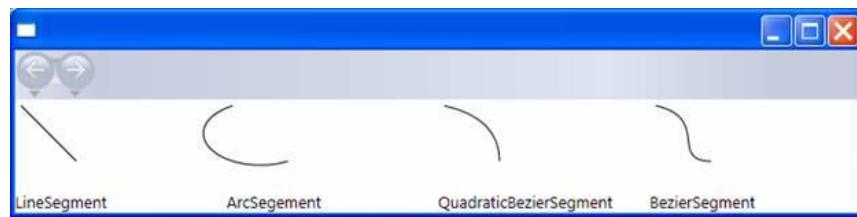


Figura 7-19 Segmentos básicos de PathGeometry

Además de estos 4 tipos básicos hay otros tres tipos de segmentos: PolyLineSegment, PolyBezierSegment y PolyQuadraticBezierSegment; que no son más que una combinación optimizada de LineSegment, QuadraticBezierSegment y BezierSegment.

En esta lección no profundizaremos en las especificidades matemáticas de cada una de estas figuras geométricas. Sólo describiremos algunas características que puedan serle útiles a Ud. para usar estas figuras.

Segmentos de línea (LineSegment): Los segmentos de línea son los más fáciles de usar: Tienen una sola propiedad adicional al resto de los segmentos: Point, que indica la posición del segundo punto de un segmento de recta. Recuerde que el primer punto de todos los segmentos es el último punto del segmento que lo antecede o el StartPoint del PathFigure.

Beziers (BezierSegment): Un bezier es una curva definida por una expresión cúbica. Matemáticamente es posible obtener una de estas curvas a partir de cuatro puntos. En BezierSegment el primero de estos puntos se define como en el resto de los segmentos, y el último punto es el que se indique en la propiedad Point3. Las propiedades Point1 y Point2 contienen los llamados puntos de control de la curva que permiten estirar los dos arcos que caracterizan a este tipo de trazo.

Beziers cuadráticos (QuadraticBezier): Un bezier cuadrático es una curva definida por una expresión cuadrática. Su dibujo es de una parábola y se matemáticamente puede obtenerse a

partir de tres puntos. El primero de estos puntos es el último punto del segmento anterior en el orden de segmentos del Path, y los dos siguientes se expresan por las propiedades Point1 y Point2. Point2 es el punto donde termina la curva y Point1 es el punto de control que determina el grado de curvatura de la parábola.

Segmento de arco elíptico (ArcSegment): Este es uno de los más empleados en la mayor parte de las figuras. Define un segmento de elipse que comienza en el punto inicial por defecto que tienen todos los segmentos y termina en el punto que se indique en la propiedad Point. Además ArcSegment cuenta con una propiedad Size cuyos valores Width y Height se interpretan como los radios horizontal y vertical respectivamente de la elipse. La mayoría de las veces estas dos propiedades son suficientes para dibujar un arco de elipse, pero ArcSegment tiene otras propiedades que enriquecen el dibujo del arco y que nos facilitan el trabajo.

Por cada par de puntos y un Size se pueden obtener muchos tipos distintos de arcos de elipse. Si los radios que se indican en el Size son mucho mayores que las dimensiones del rectángulo imaginario que se forma entre el primero y segundo punto, la elipse que se forma puede tener muchos arcos que pasen por estos puntos. Si por ejemplo los radios de la elipse son iguales a las dimensiones horizontal y vertical del rectángulo imaginario, usted puede encontrar por lo menos cuatro arcos de elipse que se puedan dibujar (Figura 7-20).

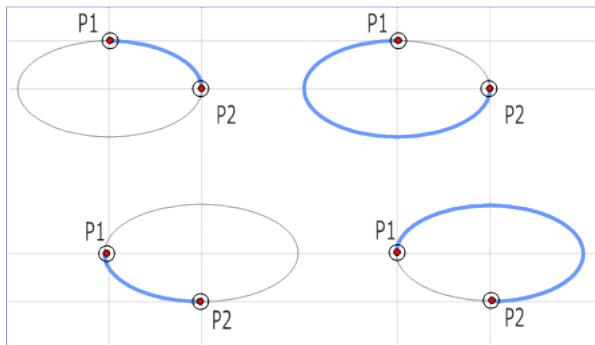


Figura 7-20 Cuatro formas de obtener un arco de una elipse determinada por radios iguales a la distancia horizontal y vertical de dos puntos P1 y P2.

Para reducir estas ambigüedades ArcSegment cuenta con las siguientes propiedades:

IsLargeArc: Permite indicar que se quiere obtener el arco largo en vez del corto. Por defecto WPF asume el corto pero usted puede seleccionar el largo asignando true a esta propiedad.

SweepDirection: Esta propiedad puede tener uno de dos valores: *Clockwise* (sentido de las manecillas del reloj) o *CounterClockwise* (sentido contrario a las manecillas del reloj).

Con estas dos propiedades podemos obtener los arcos diferentes de la Figura 7-17.

RotationAngle: Esta propiedad permite obtener una elipse que pasa por los dos puntos inicial y final y que además está rotada en un ángulo correspondiente con el valor que se indique

respecto del eje X. Esta propiedad es útil cuando las dimensiones de la elipse son mucho mayores que la distancia por la horizontal y vertical entre los puntos.

7.4.3 Dibujando un clip

Como dijimos anteriormente un Path está compuesto por una colección de segmentos que se ubican en un PathFigure. A través del dibujo de un clip ilustraremos como crear un Path compuesto por más de un segmento.

En la Lección **Documentos** se trata el tema de las anotaciones en WPF. Uno de los tipos de anotaciones más atractivo es la pegatina. En el ejemplo final de este curso usaremos también pegatinas pero suponga que queremos que las pegatinas aparezcan con un clip como se muestra en la Figura 7-21. En lo que resta de esta lección vamos a ver cómo dibujar este clip.

El Listado 7-15 nos dibuja un rectángulo que hará de pegatina, es sobre este rectángulo que vamos a poner un Path para dibujar el clip sobre su borde superior derecho. El Path estará compuesto por un segmento de arco que hará la parte más alta del clip, luego un segmento de línea que continúa con otro segmento de arco en la parte más baja, y finalmente termina el clip con otro segmento de línea que finaliza un poco más abajo del borde del rectángulo (Listado 7-16).

Listado 7-15 Página con un rectángulo amarillo claro que representa la pegatina.

```
<Page x:Class="DrawingSample.Shapes7"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Shapes7" >
    <Grid>
        <Rectangle Stroke="Silver" Fill="LightYellow"
            Margin="10,20" Height="210" Width="250"
            VerticalAlignment="Top" HorizontalAlignment="Left"/>
    </Grid>
</Page>
```

Note que hemos tenido que usar la propiedad SweepDirection de ArcSegment porque por defecto los arcos se dibujan con convexidad invertida a la que necesitamos para el clip. Note que si en lugar de empezar por el arco de más arriba de el clip, hubiésemos comenzado a dibujar por el segmento de línea con que hemos terminado, la dirección del dibujo hubiera sido favorable y no habría que cambiar el valor de SweepDirection (el resultado se muestra en la Figura 7-21).

Listado 7-16 Path que esboza un clip sobre el borde del

rectángulo

```
<Page x:Class="DrawingSample.Shapes7"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Shapes7" >
<Grid>
    <Rectangle .../>
    <Path Stroke="#444" StrokeThickness="3" Margin="20,20">
        <Path.Data>
            <PathGeometry>
                <PathFigure StartPoint="5,0">
                    <ArcSegment Size="10,18" Point="20,0"
                        SweepDirection="Clockwise"/>
                    <LineSegment Point="25,60"/>
                    <ArcSegment Size="7.5,9" Point="10,60"
                        SweepDirection="Clockwise"/>
                    <LineSegment Point="10,26"/>
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>
</Grid>
</Page>
```

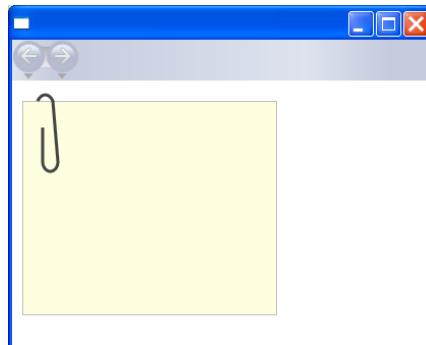


Figura 7-21 Clip dibujado sobre el borde superior del rectángulo.

Lección 8 Brochas

Además de las figuras en WPF disponemos también de brochas con las que podemos mejorar sustancialmente el aspecto de nuestras aplicaciones.

Las brochas son los recursos por excelencia para colorear las superficies y contornos de los elementos visuales. Permiten colorear el fondo y los bordes de las figuras con colores enteros, con gradientes, con imágenes e incluso con otros elementos visuales como figuras, paneles o controles.

A partir de una brocha se pueden construir también pinceles. Los pinceles son los encargados de definir cómo se dibujan las líneas y contornos de los elementos visuales, de modo que usted pueda dibujar contornos con líneas discontinuas, indicar el grosor del contorno y otras facilidades.

8.1 Brochas de Colores enteros

SolidColorBrush es la brocha más simple de WPF. Con ella se rellenan las superficies y bordes con un color entero. Tiene una propiedad Color para indicar el color con que se puede colorear.

8.1.1 Método abreviado

En las lecciones anteriores se han empleado brochas en varias ocasiones. Por ejemplo, cuando usted define el fondo de un control de la forma <Button Background="Tomato"> realmente usted está asignando una brocha de tipo SolidColorBrush a la propiedad Background del botón empleando un método abreviado. Un método abreviado es una forma de definición breve que se emplea en WPF para agilizar la escritura en XAML, que se basa en convertir, lo que se expresa originalmente en XAML como una cadena de caracteres, en un objeto del tipo apropiado para una propiedad. Así por ejemplo, una cadena como "Tomato" asignada a una propiedad de tipo Brush se convierte en una asignación de un objeto de tipo SolidColorBrush a dicha propiedad. La forma explícita de hacer esta asignación es la que se muestra en el Listado 8-1:

Listado 8-1 Asignación explícita de un SolidColorBrush a una propiedad de tipo Brush.

```
<Button>
  <Button.Background>
    <SolidColorBrush Color="Tomato"/>
  </Button.Background>
</Button>
```

Se aplica aquí un método abreviado cuando se hace <SolidColorBrush Color = "Tomato"/>, "". El método abreviado hace que WPF utilice un conversor de tipos (TypeConverter) que se le ha indicado como atributo al tipo Color y el cual convierte cadenas (como "Tomato" en este caso)

en un objeto Color. Para ello, el conversor de tipos se apoya en el método FromKnownColor del tipo Color.

En XAML todos los valores se indican como cadenas de caracteres, por lo que exceptuando las propiedades de tipo String, todas las demás propiedades necesitan realmente de un TypeConverter que convierta la cadena en un objeto del tipo de la propiedad. Esto ocurre hasta cuando se trabaja con aparentes valores numéricos. En la mayoría de las situaciones esto será transparente para Usted porque la maquinaria de WPF ya lo tiene instrumentado.

8.1.2 Formación de colores por saturación de los colores primarios

Para muchos colores existe una cadena de caracteres que WPF entiende como el nombre común del color, tal es el caso de "Tomato". Sin embargo, no es posible tener un nombre predefinido para la inmensa variedad de colores, WPF permite representar un formato particular de formación de colores, basado en la saturación de rojo, verde, azul y el grado de transparencia con que se desea crear el color. Recuerde que todo color puede ser logrado a partir de la combinación de los tres colores primarios (**Rojo, Verde y Azul**). El grado de saturación de cada uno marca la diferencia visual entre unos colores y otros. Por ejemplo el negro es ausencia total de luz y se representa como **#000000**, con saturación nula en los tres colores primarios. Por el contrario el blanco es presencia total de los tres colores primarios y se representa como **#FFFFFF**, lo cual significa que se han saturado los tres canales de colores al máximo (255).

En WPF hay varias formas de escribir estas cadenas de definición de colores. Las cadenas deben siempre empezar con el símbolo # y a continuación pueden tener 3, 4, 6 u 8 números en base Hexadecimal. Cada una de estas posibles cadenas se interpreta de la siguiente forma:

Seis números hexadecimales (#RRGGBB): Las dos primeras cifras (RR) representan un número en notación hexadecimal entre 0 y 255 que indican el grado de saturación de Rojo en el color (Red). Las siguientes dos cifras (GG) indican el grado de saturación de Verde en el color (Green) y el tercer par (BB) indica la saturación de Azul (Blue).

Tres números hexadecimales (#RGB): Es una forma aún más abreviada de indicar colores por saturación, donde las cifras se asumen repetidas. Por ejemplo al definir **#80F** como fondo de un rectángulo: <Rectangle Fill="#80F"/> estamos indicando realmente el **#8800FF** o sea un color con **88** (136 en decimal) de saturación roja, **00** (nada) de verde y **FF** (255 en hexadecimal) saturación total de azul. Con esto hemos construido un color violáceo que se muestra en la Figura 8-1.



Figura 8-1 Color #80F

Ocho números hexadecimales (#AARRGGBB): Esta es la forma más detallada de formar un color. Los colores en WPF se expresan en formato de 32 bits, o sea 4 bytes. Los seis últimos caracteres hexadecimales (...RRGGBB) que nos forman tres bytes indican la saturación de Rojo, Verde y Azul del color y el primer byte (AA...) indica que este color tiene además un grado de opacidad que WPF emplea para aplicar efectos de transparencia. Un valor **00** en el grado de opacidad indica que el color es completamente transparente (deja ver completamente lo que tenga detrás), por el contrario el valor **FF** para las cifras AA indica que el color es completamente opaco (no deja ver nada lo que tenga detrás). La Figura 8-2 muestra una elipse con un color negro semitransparente (**#80000000**) que se ha dibujado sobre una imagen.



Figura 8-2 Elipse coloreada de #80FF8000

Cuatro números hexadecimales (#ARGB): Esta es una forma abreviada del método de 8 cifras donde cada cifra se asume repetida en una cadena de 8 cifras. Por ejemplo el color **#8F80** es el equivalente al color **#88FF8800**.

En el transcurso de esta lección y del resto del curso emplearemos con mucha frecuencia esta forma de representación de colores para crear brochas particularmente útiles en la simulación de efectos de brillo y sombra y para las cuales no hay un nombre predefinido conocido por WPF.

8.2 Gradientes de colores

Las brochas del tipo `SolidColorBrush` permiten llenar fondos con un color entero lo que le da una impresión plana de las figuras que estamos coloreando. Sin embargo, las interfaces de usuario modernas emplean otros tipos de brochas que aportan una apariencia más atractiva y un aspecto más realista a los elementos que componen la interfaz. Por ejemplo, cuando a un cuerpo le da la luz, aunque esté coloreado uniformemente de un solo color, la luz es reflejada o refractada de diferentes formas acorde con el material de que esté hecho el objeto. Usted seguramente ha visto aplicaciones con botones que dan efectos cristalinos, metálicos, plásticos. Estos efectos visuales se logran en su mayor parte usando brochas que imitan la forma en que la luz incide sobre los diferentes materiales. El estudio de los efectos de luminiscencia puede llegar a ser un instrumento muy útil para lograr apariencias muy atractivas de las aplicaciones. Aunque tratar esto en detalle va más allá del objetivo de este curso, en esta sección le daremos algunos trucos para lograr efectos atractivos empleados usando las brochas de gradientes.

De manera natural la luz suele crear un brillo donde más incidencia tiene sobre el objeto y una sombra donde menos incide. Esta regla se aplica en general para casi todos los materiales opacos. Otros materiales como el cristal dejan pasar gran parte de la luz dejando a veces de forma distorsionada la imagen de los cuerpos que estén tras ellos. Aunque estos dos tipos de materiales pueden lucir bastante diferentes, ambos tienen en común que allí donde se nota el brillo de la luz, el color del cuerpo es más claro, mientras que donde aparece la sombra el color del cuerpo suele ser más oscuro, formándose entonces una degradación del color desde su variante más clara, pasando por su color original y terminando en una versión más oscura. La Figura 8-3 muestra a la izquierda un rectángulo de esquinas redondeadas coloreado donde se han aplicado efectos de brillo y sombra y otro en el que se ha simulado un efecto cristalino. WPF permite lograr estas degradaciones de colores usando brochas de gradientes: LinearGradientBrush y RadialGradientBrush.

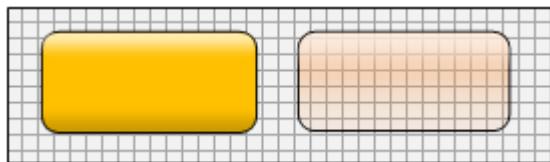


Figura 8-3 A la izquierda cuerpo opaco con brillo y sombra. A la derecha cuerpo cristalino semitransparente con poco brillo y sin sombra.

8.2.1 LinearGradientBrush

En la Lección **Figuras** se mostró un ejemplo donde se dibuja una estrella parecida a las que se emplean comúnmente para indicar grados de preferencias o favoritos en aplicaciones como el Media Player (Figura 8-4). Sin embargo, note que esta estrella se ve bastante plana. La Figura 8-5 muestra otra versión de esta misma estrella que se ha rellenado usando LinearGradientBrush.



Figura 8-4 Estrella de la lección: Figuras. Figura 8-5 Estrella coloreada con gradiente lineal.

La estrella de la Figura 8-5 se ha rellenado con un gradiente vertical que comienza en amarillo claro, pasa por amarillo puro, cambia a color dorado y termina en un dorado oscuro (Listado 8-2).

Listado 8-2 Uso del LinearGradientBrush para colorear un polígono.

<code><Page x:Class="BrushesAndPencilSamples.Star"</code>
--

```

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="BrushesAndPencilSamples" Height="300" Width="300" >
<Grid>
    <Polygon Points="10,40 32,40 40,10 48,40 70,40 53,53 60,80 40,65 20,80
27,53"
        Stroke="DarkGoldenrod">
        <Polygon.Fill>
            <LinearGradientBrush EndPoint="0,1">
                <GradientStop Color="LightYellow" Offset="0"/>
                <GradientStop Color="Yellow" Offset="0.4"/>
                <GradientStop Color="Gold" Offset="0.6"/>
                <GradientStop Color="Goldenrod" Offset="1"/>
            </LinearGradientBrush>
        </Polygon.Fill>
    </Polygon>
</Grid>
</Page>

```

Dirección del gradiente

En el Listado 8-2 presentamos la brocha vertical que se emplea para llenar la Figura 8-5. ¿Pero qué hace que el gradiente sea vertical? El primer aspecto a definir cuando se usan los gradientes lineales es su dirección. LinearGradientBrush rellena siempre toda la superficie de la figura a la que se le aplica con una dirección que se calcula a partir de las propiedades **StartPoint** (con valor **0,0** por defecto) y **EndPoint**. Ambas propiedades son de tipo Point y los valores que deben tomar son pares de números reales entre 0 y 1, donde (0,0) indica la esquina superior derecha de un rectángulo imaginario que enmarca a la figura, y (1,1) representa la esquina inferior derecha de este rectángulo (Figura 8-6).

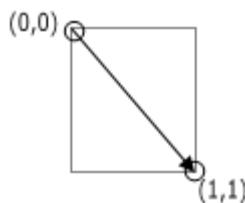


Figura 8-6 Posiciones (0,0) y (1,1) en el rectángulo imaginario que LinearGradientBrush usa para calcular el área a colorear

En el LinearGradientBrush del Listado 8-2 hemos indicado que el punto final es **(0,1)**. Como el punto inicial es por defecto **(0,0)** la dirección del gradiente desde **(0,0)** hasta **(0,1)** define una dirección vertical. La Figura 8-7 muestra la dirección del gradiente aplicado a la estrella.

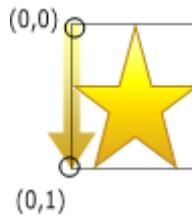


Figura 8-7 Dirección vertical de un gradiente lineal.

Ubicación de los colores en el gradiente

El otro elemento de relevancia en las brochas de gradientes es la ubicación de los diferentes colores que forman al gradiente. Los GradientStops son los elementos que permiten indicar cuáles son los colores que forman al gradiente y en qué posición deben aparecer respecto del rectángulo que enmarca a la figura. Cada GradientStop tiene una propiedad Color donde se indica el color y una propiedad Offset que debe tomar un valor entre 0 y 1 e indica la posición relativa a partir de la cual la brocha tomará ese color en correspondencia con el largo del gradiente. En el ejemplo anterior (Listado 8-2) hemos puesto cuatro colores: El primero (**LightYellow**) en la posición 0 que en el gradiente vertical colorea la parte más alta de la figura. El segundo (**Yellow**) lo hemos puesto en la posición 0.4, que en el gradiente vertical corresponde con el 40% de la altura de la figura. Un tercer color (**Gold**) está puesto en 0.6 o 60% de la altura de la figura y finalmente, hemos puesto de color dorado oscuro (**Goldenrod**) el extremo más bajo de la figura. Pruebe a crear un efecto arco iris!

Brillo y sombra

El efecto de brillo y sombra sobre una figura se obtiene combinando convenientemente brochas de color entero y brochas de gradientes basados en colores semitransparentes. Para lograr un efecto como el del primer rectángulo con esquinas redondeadas de la Figura 8-3, comenzaremos por dibujar un borde coloreado enteramente de color dorado (Listado 8-3 y Figura 8-8).

Listado 8-3 Definición de un borde relleno de dorado con esquinas redondeadas.

```
<Page x:Class="BrushesAndPencilSamples.BrightAndShadow1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BrightAndShadow1" >
    <Canvas>
        <Border CornerRadius="10" Width="120" Height="70"
            Background="Gold" BorderBrush="DarkGoldenrod" BorderThickness="2"
            Canvas.Left="10" Canvas.Top="10"/>
    </Canvas>
</Page>
```

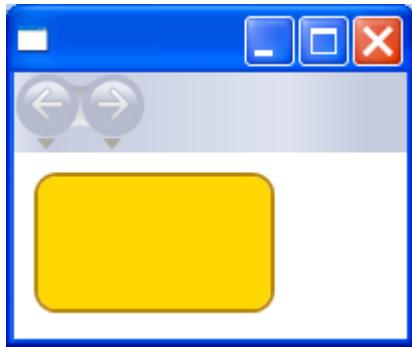


Figura 8-8 Borde relleno de color dorado.

Dentro de este borde pondremos otro con un relleno diferente. Ahora utilizaremos como relleno un gradiente vertical basado en colores semitransparentes que combinados con el dorado de fondo provoquen los efectos de brillo y sombra. Como dijimos con anterioridad el brillo se logra aclarando el color del cuerpo iluminado, esto podemos lograrlo indicando que el primer color del gradiente será un blanco con cierto grado de transparencia, por ejemplo: #CFFF. Al aplicar este color WPF combina el color de fondo (**Gold**) con el color blanco (...FFF) de forma que el color que se ve en la pantalla es realmente un intermedio entre ambos, que estará más cercano a uno que al otro en correspondencia con la opacidad que pongamos (#C...). Luego dejaremos que el color de fondo se vea tal cual cubriendo la mayor parte de la superficie del borde, poniendo al color transparente (**Transparent**) en las posiciones **0.2** y **0.8** del gradiente. Finalmente, para lograr el efecto de sombra, pondremos como último color del gradiente un negro casi transparente (#4000) que combine bien con el dorado para lograr un efecto de obscurecimiento en la parte inferior de la figura (Listado 8-4 y Figura 8-9).

Listado 8-4 Borde interno que simula los efectos de sombra y brillo

```
<Page ...>
<Canvas>
    <Border CornerRadius="10" Width="120" Height="70"
        Background="Gold" BorderBrush="DarkGoldenrod"
        BorderThickness="2"
        Canvas.Left="10" Canvas.Top="10">
        <Border CornerRadius="6">
            <Border.Background>
                <LinearGradientBrush EndPoint="0,1">
                    <GradientStop Color="#CFFF" Offset="0"/>
                    <GradientStop Color="Transparent" Offset="0.2"/>
                    <GradientStop Color="Transparent" Offset="0.8"/>
                    <GradientStop Color="#4000" Offset="1"/>
                </LinearGradientBrush>
            </Border.Background>
        </Border>
    </Border>
</Canvas>
```

```
</Border>  
</Border>  
</Canvas>  
</Page>
```

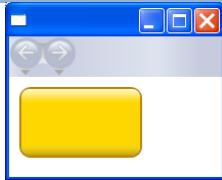


Figura 8-9 Borde relleno con gradiente vertical de colores blanco y negro semitransparentes.

Compare usted la Figura 8-8 y la Figura 8-9, y note como en los efectos logrados con el gradiente de colores semitransparentes le dan cuerpo a la figura plana que se tenía al principio.

Cómo lograr un efecto cristalino como el del segundo rectángulo de la Figura 8-3 lo veremos en la Lección **Efectos visuales** cuando se expliquen otros efectos de dibujo.

8.2.2 RadialGradientBrush

El RadialGradientBrush es también una brocha muy útil para lograr efectos de iluminación. La Figura 8-10 muestra una ventana donde se ha coloreado el fondo con un gradiente radial para resaltar un área de la misma.



Figura 8-10 Fondo rellenado con una brocha radial.

El Listado 8-9 corresponde con la Figura 8-10 al cual llegaremos analizando paso a paso las propiedades de esta brocha.

Ubicación de los colores en el gradiente

Al igual que la brocha lineal, la brocha de gradiente radial cuenta con una colección de GradientStops que permiten indicar los colores que conforman el gradiente. En RadialGrandientBrush el color que se encuentra en la posición 0 es el que se dibuja en el centro, los demás colores se dibujan formando anillos hasta el borde más externo, el cual se rellena con el color de la posición 1. La Figura 8-11 muestra el modo en que se dibujan los colores que forman un RadialGradientBrush que pasa del rojo, al amarillo y al azul. (Listado 8-5).

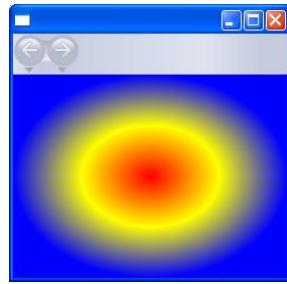


Figura 8-11 Grid coloreado con un gradiente radial de tres colores: Rojo, Amarillo y Azul.

Listado 8-5 Definición de una brocha de gradiente radial como relleno de un Grid

```
<Page x:Class="BrushesAndPencilSamples.RadialPage2"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="RadialPage2" >
  <Grid>
    <Grid.Background>
      <RadialGradientBrush>
        <GradientStop Color="Red" Offset="0"/>
        <GradientStop Color="Yellow" Offset="0.5"/>
        <GradientStop Color="Blue" Offset="1"/>
      </RadialGradientBrush>
    </Grid.Background>
  </Grid>
</Page>
```

Note que hemos puesto al amarillo entre el rojo y el azul (`Offset="0.5"`). Si en vez de esto hacemos que el amarillo aparezca al 65% del radio del gradiente (`Offset="0.65"`) y el azul al 70% (`Offset="0.7"`) fíjese en la Figura 8-12 como el color rojo del centro se extiende más y cómo se atenúa la separación entre el amarillo y el azul.

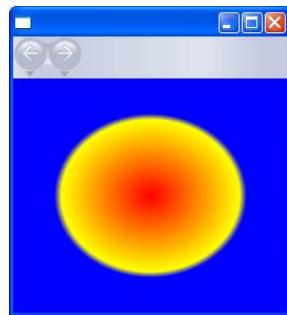


Figura 8-12 Color Amarillo en la posición 0.65 de un RadialGradientBrush.

Usted habrá notado también que los colores rojo y amarillo están ahora más separados de los bordes de la ventana. Esto se debe a que el azul está en una posición equivalente al 70% (`Offset="0.7"`) de la distancia entre el centro del Grid y sus bordes. La brocha toma como último

color al que tenga una posición más cercana a 1, y con este color completa el relleno de toda el área a colorear.

Centro

En un RadialGradientBrush hay que poder ubicar dónde está el centro, para ello se tiene la propiedad Center de tipo Point que debe tomar valores entre 0 y 1 relativos a las dimensiones del rectángulo imaginario que contiene a la figura que se está llenando con esta brocha. Por defecto el valor de esta propiedad es **0.5,0.5** lo que ubica al gradiente en el centro de la figura. Si cambiáramos este valor por **0.3,0.4** (Listado 8-6) se produce el efecto de la Figura 8-13.

Listado 8-6 Brocha de gradiente radial con el centro cambiado

```
<Page ...>
<Grid>
    <Grid.Background>
        <RadialGradientBrush Center="0.3,0.4">
            <GradientStop Color="Red" Offset="0"/>
            <GradientStop Color="Yellow" Offset="0.5"/>
            <GradientStop Color="Blue" Offset="1"/>
        </RadialGradientBrush>
    </Grid.Background>
</Grid>
</Page>
```

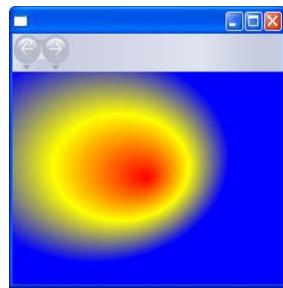


Figura 8-13 Grid llenado con una Brocha de gradiente radial con centro en 0.3, 0.4

Aunque no estamos usando explícitamente una elipsis observe que toda el área que no es de color azul entero y advertirá que efectivamente tiene forma elíptica. El cambio de la propiedad Center provoca este efecto de corrimiento porque RadialGradientBrush calcula una elipse que se ubica en el nuevo centro con dimensiones iguales a las del elemento a llenar (toda la celda del Grid en este caso). Es en el interior de esta elipse donde se aplica el gradiente de colores. Fuera de esta área se rellena exclusivamente con el color que esté en la última posición del gradiente.

Pero volvamos a echar un vistazo a la Figura 8-13. Note que el color rojo sigue estando en el centro de la celda del grid y no en el centro de los efectos de gradiente que se ha desplazado. El corrimiento del centro de la brocha no afecta al origen desde donde se comienza a colorear el gradiente, sino que permanece en la posición original (0.5, 0.5). Si quisieramos que el centro desde dónde se empieza a colorear coincidiera con el centro del gradiente (la elipse desplazada), tendríamos que correr también el origen del gradiente que se corresponde con la propiedad GradientOrigin (Listado 8-7 y Figura 8-14).

Listado 8-7 Centro y origen de gradiente en la misma posición de una brocha de gradiente radial

```
<Page ...>
  <Grid>
    <Grid.Background>
      <RadialGradientBrush Center="0.3,0.4"
        GradientOrigin="0.3,0.4">
        ...
      </RadialGradientBrush>
    </Grid.Background>
  </Grid>
</Page>
```

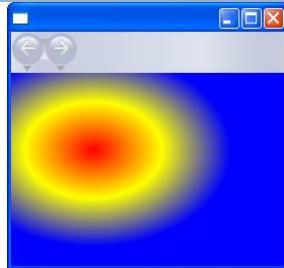


Figura 8-14 Grid llenado con una brocha de gradiente radial con centro y origen de gradiente en la misma posición.

Dimensiones

Para completar nuestras expectativas respecto del RadialGradientBrush sería deseable poder extender la elipse a dimensiones mayores a las que tiene originalmente, para poder completar toda el área de relleno. Las propiedades RadiusX y RadiusY de RadialGradientBrush nos permiten cambiar las dimensiones horizontales y verticales de la elipse imaginaria que esta brocha calcula para llenar. Por defecto estas dos propiedades tienen un mismo valor 0.5 relativo a las dimensiones horizontal y vertical del elemento a llenar.

En el ejemplo anterior hemos corrido el centro y origen de gradiente a **0.3** del ancho del Grid y a **0.4** de su altura. Si completáramos el área restante dándole valores superiores: **1.1** a RadiusX y **0.9** a RadiusY (Listado 8-8) podemos entonces hacer crecer la elipse rojo-amarilla a dimensiones mayores que las del Grid (Figura 8-15).

Listado 8-8 Brocha de gradiente radial con radios aumentados

```
<Page ...>
<Grid>
    <Grid.Background>
        <RadialGradientBrush Center="0.3,0.4"
GradientOrigin="0.3,0.4" RadiusX="1.1" RadiusY="0.9">
        ...
    </RadialGradientBrush>
</Grid.Background>
</Grid>
</Page>
```

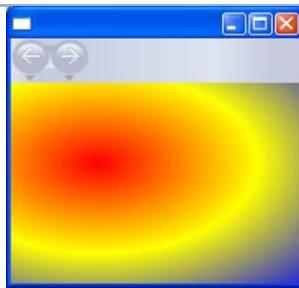


Figura 8-15 Grid coloreado con brocha de gradiente radial con radios aumentados a 1.1 y 0.9

Ahora ya podemos entender con el (Listado 8-9) cómo se logró el efecto que mostramos originalmente con la Figura 8-10. Se pone un fondo de relleno entero para la ventana con azul marino (**Navy**) y luego ponemos un Grid encima relleno con un gradiente radial que aprovecha la combinación de colores semitransparentes. Note que el centro del gradiente está ubicado en la 0.3, 0.4 y que el texto **Brochas** está ubicado a un margen 30, 40, esto es lo que nos da el efecto de una iluminación que rodea al texto.

Listado 8-9 XAML de la Figura 8-10

```
<Page x:Class="BrushesAndPencilsSamples.RadialPage1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="RadialPage1" Background="Navy" >
    <Grid>
        <Grid.Background>
            <RadialGradientBrush Center="0.3,0.4" GradientOrigin="0.2,0.3"
RadiusX="1" RadiusY="1">
                <GradientStop Color="#c0af" Offset="0"/>
                <GradientStop Color="#608f" Offset="0.4"/>
                <GradientStop Color="Transparent" Offset="0.7"/>
            </RadialGradientBrush>
        </Grid.Background>
        <TextBlock FontSize="30" Margin="30,40" Foreground="White"
VerticalAlignment="Top" HorizontalAlignment="Left">
```

```
    Brochas  
  </TextBlock>  
</Grid></Page>
```

8.3 Tapizando tu fondo (TileBrush)

Hasta aquí hemos visto como "pintar" de colores nuestros fondos, vamos a ver ahora cómo también lo podemos "tapizar". En la Figura 8-3 de esta lección usted habrá notado que hemos dibujado los rectángulos sobre una superficie que simula un trozo de hoja cuadriculada. Este efecto cuadriculado se ha logrado empleando un tipo de brocha que repite un determinado dibujo sobre la superficie (como si fuera un tapiz sobre una pared). En WPF se puede lograr esto con tres brochas que heredan de TileBrush: ImageBrush, VisualBrush y DrawingBrush. En la sección a continuación estudiaremos primero a ImageBrush para mostrar las particularidades de todas las TileBrush y luego en la siguiente sección explicaremos cómo lograr el efecto de hoja cuadriculada con DrawingBrush.

8.3.1 ImageBrush

Como su nombre lo indica ImageBrush es la brocha que nos permite rellenar un fondo con una imagen. Recordemos que podemos poner un elemento Image dentro de otro elemento, pero Image es propiamente un elemento de interacción que recibe eventos de teclado y ratón y que además se puede ajustar al tamaño del elemento contenedor siguiendo todas las bondades de la alineación y margen. El propósito de ImageBrush es otro, con ImageBrush se puede llenar el fondo completo con una imagen sin que esto signifique que se ha puesto una imagen como un elemento de interacción. Además ImageBrush hereda de TileBrush otras bondades que se describirán en esta sección. La Figura 8-16 muestra una página cuyo fondo se ha llenado con la imagen de Windito (Listado 8-10).



Figura 8-16 Página con una imagen de fondo.

Listado 8-10 Fondo de página relleno con un ImageBrush

```
<Page x:Class="BrushesAndPencilSamples.ImageBrush1"  
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
      Title="ImageBrush1" >  
<Page.Background>
```

```
<ImageBrush ImageSource="Images/Windito.bmp"/>
</Page.Background>
</Page>
```

Las propiedades que a continuación describiremos son aplicables a ImageBrush y las demás brochas herederas de TileBrush.

Extensión de la imagen dentro del área (Stretch)

Si ahora redimensionáramos la ventana de la Figura 8-16 notaremos como la figura también se redimensiona acorde con el nuevo tamaño de la ventana (Figura 8-17).



Figura 8-17 Ventana de la Figura 8-16 redimensionada

Fill: El efecto de la Figura 8-17 ocurre porque TileBrush tiene una propiedad Stretch que indica cuándo y cómo el contenido de la brocha debe extenderse para ajustarse al tamaño del elemento que la brocha está rellenando. Por defecto esta propiedad tiene valor Fill que indica que la imagen debe cubrir toda el área del elemento, pero Stretch puede tomar también los valores que se exponen a continuación y que usted puede detallar en la Figura 8-18 y Listado 8-11.

None: La imagen no se extiende en lo absoluto y se muestra con su tamaño original. La Figura 8-18 muestra como se rellenaría el fondo de la página con Stretch="None". Por defecto además de mantener su tamaño original la imagen queda centrada. En la siguiente sección veremos cuales propiedades están involucradas con esto.

UniformToFill: El valor UniformToFill en la propiedad Stretch hace que la imagen se extienda vertical y horizontalmente para ajustarse al tamaño de la página sin perder sus proporciones. Note en la Figura 8-18 que la imagen se redimensiona cubriendo el fondo del rectángulo.

Uniform: Con Uniform la imagen se ajusta a la menor de las dimensiones vertical u horizontal del área de relleno de modo que la imagen siempre aparece completamente contenida dentro del área de relleno sin perder sus proporciones. La diferencia entre Uniform y UniformToFill se hace aún más evidente cuando las proporciones del área de relleno no coinciden con las de la

imagen. Ambos mantienen las proporciones de la imagen pero como con Uniform debe garantizarse que la imagen se vea completamente en ocasiones queda una porción del área de relleno sin cubrir. Observe en la Figura 8-18 como al llenar el rectángulo con extensión Uniform han quedado dos espacios sin llenar a ambos lados de la imagen.

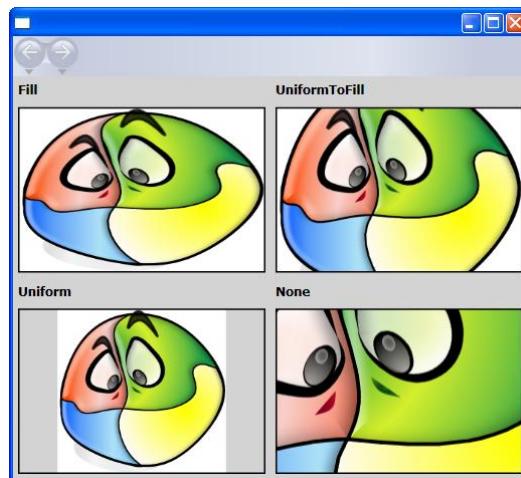


Figura 8-18 Efecto de los cuatro valores que puede tomar la propiedad Stretch de ImageBrush

Listado 8-11 Varios rectángulos llenados con la misma imagen pero con diferentes formas de extensión

```
<Page x:Class="BrushesAndPencilsSamples.ImageBrush1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="ImageBrush1" Background="LightGray" >
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <TextBlock FontWeight="Bold" Margin="5">
    Fill
  </TextBlock>
  <Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Row="1" Margin="5">
    <Rectangle.Fill>
      <ImageBrush ImageSource="Images/Windito.bmp"/>
    </Rectangle.Fill>
  </Rectangle>
  <TextBlock FontWeight="Bold" Margin="5" Grid.Row="2">
    Uniform
  </TextBlock>
</Grid>
```

```

</TextBlock>
<Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Row="3" Margin="5">
<Rectangle.Fill>
<ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"/>
</Rectangle.Fill>
</Rectangle>
<TextBlock FontWeight="Bold" Margin="5" Grid.Column="1">
    UniformToFill
</TextBlock>
<Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Row="1" Grid.Column="1"
    Margin="5">
<Rectangle.Fill>
<ImageBrush ImageSource="Images/Windito.bmp" Stretch="UniformToFill"/>
</Rectangle.Fill>
</Rectangle>
<TextBlock FontWeight="Bold" Margin="5" Grid.Column="1" Grid.Row="2">
    None
</TextBlock>
<Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Column="1" Grid.Row="3"
    Margin="5">
<Rectangle.Fill>
<ImageBrush ImageSource="Images/Windito.bmp" Stretch="None"/>
</Rectangle.Fill>
</Rectangle>
</Grid>
</Page>

```

Alineación

En todas las figuras de la sección anterior hemos visto cómo la imagen aparece alineada en el centro de la página. En la Figura 8-19 se muestra la misma imagen alineada a la izquierda usando la propiedad `AlignmentX` como se muestra en el Listado 8-12.



Figura 8-19 Fondo relleno con una imagen alineada a la izquierda.

Listado 8-12 ImageBrush que alinea la imagen a la izquierda

```

<Page x:Class="BrushesAndPencilSamples.ImageBrush1"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ImageBrush1" Background="Tomato" >
<Grid>

```

```

<Grid.Background>
    <ImageBrush      ImageSource="Images/Windito.bmp"
    Stretch="Uniform"
        AlignmentX="Left"/>
</Grid.Background>
</Grid>
</Page>

```

La propiedad `AlignmentX` permite alinear la imagen al centro (`Center`), a la izquierda (`Left`) o a la derecha (`Right`). Así mismo `TileBrush` brinda la propiedad análoga `AlignmentY` que alinea la figura hacia el borde superior (`Top`), centro (`Center`) o pegada al borde inferior (`Bottom`).

Distribución

La distribución es la principal funcionalidad que aporta la clase base `TileBrush` a sus brochas herederas. Las propiedades relacionadas con distribuir el contenido de la brocha en el área de relleno permiten seleccionar una porción del contenido y extender este contenido, alinearlo, o repetirlo de diferentes maneras en una porción del área de dibujo. Las propiedades que permiten lograr todo esto son:

`Viewbox` y `ViewboxUnits`: Son las propiedades que permiten seleccionar una porción del contenido para aplicar la extensión alineación y relleno dentro del área de dibujo. `Viewbox` describe un rectángulo que representa el área de dibujo a tomar para extender y alinear la imagen (Figura 8-20), mientras que `ViewboxUnits` indica cómo deben interpretarse las coordenadas que se empleen para definir el `Viewbox`. Si usted conoce las dimensiones reales de la imagen, por ejemplo 588x589 el `Viewbox` de la Figura 8-20 usted puede definirlo como 83.7,95.13,223.2,144.96, indicándole al `ImageBrush` que usted está empleando coordenadas absolutas (`ViewboxUnits="Absolute"`)

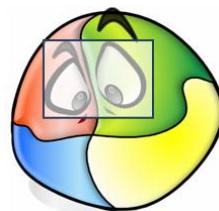


Figura 8-20 Representación del Viewbox en una imagen.

En el Listado 8-13 hemos definido un `Viewbox` con estas coordenadas y el resultado se muestra en la Figura 8-21.

Listado 8-13 ImageBrush con un Viewbox definido con coordenadas absolutas

```

<Page x:Class="BrushesAndPencilSamples.ImageBrush2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="ImageBrush2" Background="LightGray" >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBlock Margin="5">
        Viewbox absoluto: <Bold>83.7,95.13,223.2,144.96</Bold>
    </TextBlock>
    <Border Margin="5" BorderBrush="Black" BorderThickness="1.5"
Grid.Row="1">
        <Border.Background>
            <ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
Viewbox="83.7,95.13,223.2,144.96" ViewboxUnits="Absolute"/>
        </Border.Background>
    </Border>
    <TextBlock Margin="5" Grid.Row="2">
        Viewbox relativo: <Bold>0.18,0.21,0.48,0.32</Bold>
    </TextBlock>
    <Border Margin="5" BorderBrush="Black" BorderThickness="1.5"
Grid.Row="3">
        <Border.Background>
            <ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
Viewbox="0.18,0.21,0.48,0.32"
ViewboxUnits="RelativeToBoundingBox"/>
        </Border.Background>
    </Border>
    <TextBlock Grid.Row="4" Margin="5">
        Con Viewport relativo: <Bold>0.25,0.1,0.25,0.8</Bold>
    </TextBlock>
    <Border Margin="5" BorderBrush="Black" BorderThickness="1.5"
Grid.Row="5">
        <Border.Background>
            <ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
Viewbox="0.18,0.21,0.48,0.32" ViewboxUnits="RelativeToBoundingBox"
Viewport="0.25,0.1,0.25,0.8"
ViewboxUnits="RelativeToBoundingBox"/>
        </Border.Background>
    </Border>
</Grid>
</Page>

```

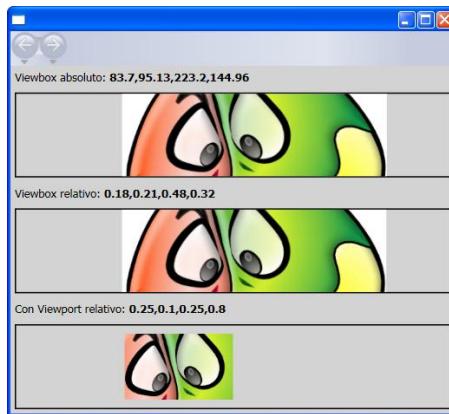


Figura 8-21 Fondo lleno con una imagen con Viewbox extendido uniformemente

Note como en la Figura 8-21 el ImageBrush en vez de extender toda la imagen para cumplir con el valor Uniform de Stretch, ha tomado como área de referencia para extender la porción de imagen seleccionada con el Viewbox.

La propiedad ViewboxUnits puede tomar también el valor: RelativeToBoundingBox, que nos sirve para indicar coordenadas relativas cuando no conocemos las dimensiones reales de la imagen. En el Listado 8-13 la brocha del segundo Border tiene definido un ViewBox relativo. Compare el primero y segundo Border y compruebe que son idénticos. La propiedad ViewboxUnits por defecto tiene valor RelativeToBoundingBox, y Viewbox tiene como valor por defecto: 0,0,1,1, de modo que la selección predeterminada coincide con las dimensiones del contenido de la brocha.

Viewport y ViewportUnits: Estas dos propiedades permiten seleccionar una porción del área sobre la cual se va a dibujar el contenido de la brocha. TileBrush toma el contenido de la brocha y lo extiende y alinea acorde con las coordenadas del Viewbox de modo que el contenido quede siempre dentro del área definida por el Viewport. En la Figura 8-21 se muestra un tercer Border lleno con la misma brocha pero con un Viewport ubicado en la segunda cuarta parte del área horizontal de la ventana y con un margen superior de un 0.1 del área vertical y con un tamaño correspondiente al 0.8 del tamaño vertical, lo cual deja un margen inferior también de 0.1.

El valor predeterminado de ViewportUnits es RelativeToBoundindBox y el de Viewport es 0,0,1,1.

¿Y qué hacemos entonces con el espacio que queda vacío? Para completar el área, TileBrush tiene una propiedad TileMode que sirve de colofón a todas estas propiedades de distribución.

Relleno

TileMode es la propiedad encargada de llenar todo el espacio que queda vacío fuera del Viewport, y emplea al propio Viewport para llenarlo. TileMode tiene 5 valores posibles que indican las diferentes maneras de llenar el espacio sobrante:

None: No rellena en lo absoluto el espacio sobrante.

Tile: Repite el Viewport en todo el espacio sobrante logrando el efecto tapizado (Listado 8-14 y Figura 8-22).

Listado 8-14 ImageBrush con efecto tapizado

```
<Page x:Class="BrushesAndPencilSamples.ImageBrush3"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ImageBrush3" Background="LightGray" >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Margin="5">
        <Bold>Tile</Bold> con coordenadas relativas
    </TextBlock>
    <Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Row="1" Margin="5">
        <Rectangle.Fill>
            <ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
                TileMode="Tile"
                Viewbox="0.18,0.21,0.48,0.32" ViewboxUnits="RelativeToBoundingBox"
                Viewport="0.25,0.1,0.25,0.8" ViewportUnits="RelativeToBoundingBox"/>
        </Rectangle.Fill>
    </Rectangle>
    <TextBlock Margin="5" Grid.Column="1">
        <Bold>Tile</Bold> con coordenadas absolutas
    </TextBlock>
    <Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Row="1" Grid.Column="1"
        Margin="5">
        <Rectangle.Fill>
            <ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
                TileMode="Tile"
                Viewbox="0.18,0.21,0.48,0.32" ViewboxUnits="RelativeToBoundingBox"
                Viewport="0,0,60,60" ViewportUnits="Absolute"/>
        </Rectangle.Fill>
    </Rectangle>
    <TextBlock Margin="5" Grid.Row="2">
        <Bold>FlipXY</Bold> con coordenadas relativas
    </TextBlock>
    <Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Row="3" Margin="5">
        <Rectangle.Fill>
```

```

<ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
    TileMode="FlipXY"
    Viewbox="0.18,0.21,0.48,0.32" ViewboxUnits="RelativeToBoundingBox"
    Viewport="0.25,0.1,0.25,0.8" ViewportUnits="RelativeToBoundingBox"/>
</Rectangle.Fill>
</Rectangle>
<TextBlock Margin="5" Grid.Column="1" Grid.Row="2">
    <Bold>FlipXY</Bold> con coordenadas absolutas
</TextBlock>
<Rectangle Stroke="Black" StrokeThickness="1.5" Grid.Column="1" Grid.Row="3"
Margin="5">
    <Rectangle.Fill>
        <ImageBrush ImageSource="Images/Windito.bmp" Stretch="Uniform"
            TileMode="FlipXY"
            Viewbox="0.18,0.21,0.48,0.32" ViewboxUnits="RelativeToBoundingBox"
            Viewport="0,0,60,60" ViewportUnits="Absolute"/>
    </Rectangle.Fill>
</Rectangle>
</Grid>
</Page>

```

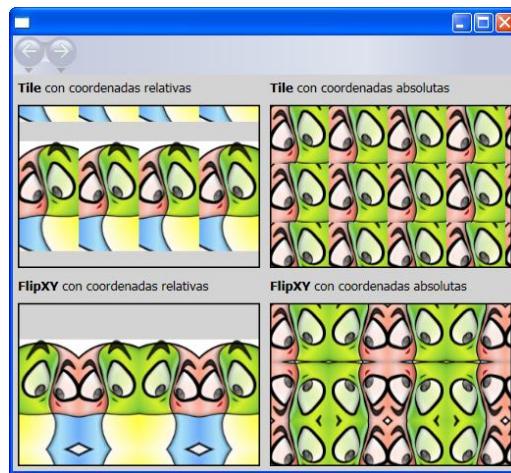


Figura 8-22 Fondo relleno con los diferentes modos de tapizado

Note como se respeta siempre el tamaño relativo del Viewport. Para obtener un fondo con más imágenes usted puede cambiar las coordenadas del Viewport a coordenadas absolutas como se muestra en la Figura 8-22.

FlipX, FlipY y FlipXY: Estas son las otras tres formas de llenar el espacio sobrante y se basan en reflejar horizontalmente, verticalmente o en ambos sentidos el contenido del Viewport durante la repetición. En el Listado 8-14 se usa FlipXY para lograr el fondo de los rectángulos que están en la segunda fila del Grid en la Figura 8-22. Observe con detenimiento y notará que en cada fila y columna la imagen ha sido reflejada como en un espejo.

8.3.2 DrawingBrush

DrawingBrush es muy útil para lograr efectos de rellenos tramados con figuras geométricas, tal es el caso del mencionado ejemplo para lograr un fondo que parezca una hoja cuadriculada. En la Figura 8-23 mostramos una ventana cuyo fondo se ha llenado usando un DrawingBrush que emplea rectángulos para "tapizar" el fondo.

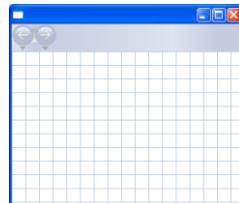


Figura 8-23 Fondo que simula una hoja cuadriculada.

La forma mas sencilla de lograr la Figura 8-23 parece ser llenar el fondo con rectángulos coloreados de blanco y bordes azul-acero claro (LightSteelBlue), pero también puede lograrse el mismo efecto si rellenáramos todo el fondo de la ventana de color LightSteelBlue para luego poner otro elemento (un Grid por ejemplo) cuyo Background sea un DrawingBrush que rellene con rectángulos blancos separados entre sí. El Listado 8-15 describe esta forma de hacerlo.

Listado 8-15 Grid lleno con un DrawingBrush que azuleja el fondo con rectángulos separados entre sí

```
<Page x:Class="BrushesAndPencilSamples.DrawingBrush1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DrawingBrush1"
    Background="LightSteelBlue" >
    <Grid>
        <Grid.Background>
            <DrawingBrush Viewbox="0,0,20,20" ViewboxUnits="Absolute"
TileMode="Tile"
                Viewport="0,0,20,20" ViewportUnits="Absolute">
                <DrawingBrush.Drawing>
                    <GeometryDrawing Brush="White">
                        <GeometryDrawing.Geometry>
                            <RectangleGeometry Rect="1,1,19,19"/>
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingBrush.Drawing>
            </DrawingBrush>
        </Grid.Background>
    </Grid>
</Page>
```

Al igual que ImageBrush, DrawingBrush es también un tipo heredero de TileBrush, y como tal tiene también las propiedades Viewbox, ViewboxUnits, Viewport, ViewporUnits, Stretch, AlignmentX, AlignmentY y Tile. Note que en el Listado 8-18 con la combinación de valores

ViewboxUnits="Absolute" y Viewbox="0,0,20,20" hemos descrito una figura de tamaño 20 x 20 que repetiremos con las mismas dimensiones (Viewport="0,0,20,20", ViewportUnits="Absolute") para tapizar (TileMode="Tile") el fondo del Grid.

DrawingBrush además cuenta con una propiedad Drawing donde describimos cuál o cuales serán los elementos de dibujo que describen la decoración. En este ejemplo nos basta con definir un dibujo geométrico (GeometryDrawing) coloreado de blanco en cuya propiedad Geometry hemos indicado que la figura geométrica a emplear será un rectángulo (RectangleGeometry). Fíjese en las dimensiones de este rectángulo: En vez de cubrir toda el área (0,0,20,20), hemos indicado que el rectángulo está separado de sus bordes superior e izquierdo en 1 pixel (1,1,19,19). Recuerde que las dos últimas dimensiones de la propiedad Rect (...,...,19,19) se refieren al tamaño del rectángulo. La Figura 8-24 muestra cómo se vería un rectángulo solo.



Figura 8-24 Representación del elemento de tapiz solo.

El área que no queda cubierta por la figura WPF la toma como transparente. Note cómo esta bondad de DrawingBrush se aprovecha en el fondo cuadriculado de la Figura 8-25 (Listado 8-16) rellenando el fondo de la ventana con una brocha de gradiente lineal diagonal. Compare las esquinas superior izquierda e inferior derecha y note los colores con los que se ha conformado el gradiente.



Figura 8-25 Grid lleno con rectángulos distribuidos sobre un fondo con gradiente lineal diagonal.

Listado 8-16 Página con relleno de gradiente lineal tapizado con rectángulos separados.

```
<Page ...>
<Page.Background>
  <LinearGradientBrush EndPoint="1,1">
    <GradientStop Color="SteelBlue" Offset="0"/>
    <GradientStop Color="White" Offset="1"/>
  </LinearGradientBrush>
```

```

</Page.Background>
<Grid>
<Grid.Background>
<DrawingBrush ...>
<DrawingBrush.Drawing>
<GeometryDrawing Brush="#a000">
...
</GeometryDrawing>
</DrawingBrush.Drawing>
</DrawingBrush>
</Grid.Background>
</Grid>
</Page>

```

Veamos ahora otro ejemplo que rellena el fondo con una brocha construída a partir de figuras geométricas. La Figura 8-26 nos muestra cómo se ve el área de visualización de videos de la aplicación **Visor de Cursos** que usamos como ejemplo general para este curso. Centrémonos primero en el fondo del área superior, donde se muestran los videos. Esta no es una imagen tomada de un archivo sino que se ha construida con un DrawingBrush, a partir de un rectángulo que cubre toda el área y que se rellena con una brocha lineal (diagonal: EndPoint="1,1") y de un fragmento de elipse que se rellena con una brocha radial.



Figura 8-26 Área de visualización de vídeos de la aplicación Visor de Cursos

Veamos ahora paso a paso cómo podemos construir un fondo como el de la Figura 8-26. Comencemos dibujando las figuras. Como son dos: un rectángulo y una elipse, utilizaremos un grupo de dibujo (ver Lección **Figuras**) que contenga estos dos elementos geométricos (Listado 8-17 y Figura 8-27).

Listado 8-17 Fondo relleno con brocha de dibujo

```

<Page x:Class="BrushesAndPencilSamples.VideoBackground"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="VideoBackground">
<Page.Background>
<DrawingBrush Viewbox="0,0,1,1" ViewboxUnits="RelativeToBoundingBox"
              Viewport="0,0,1,1" ViewportUnits="RelativeToBoundingBox">

```

```

<DrawingBrush.Drawing>
  <DrawingGroup>
    <GeometryDrawing Brush="Green">
      <GeometryDrawing.Geometry>
        <RectangleGeometry Rect="0,0,1,1"/>
      </GeometryDrawing.Geometry>
    </GeometryDrawing>
    <GeometryDrawing Brush="Blue">
      <GeometryDrawing.Geometry>
        <EllipseGeometry Center="0.5,0.5" RadiusX="0.5" RadiusY="0.35"/>
      </GeometryDrawing.Geometry>
    </GeometryDrawing>
  </DrawingGroup>
</DrawingBrush.Drawing>
</DrawingBrush>
</Page.Background>
</Page>

```

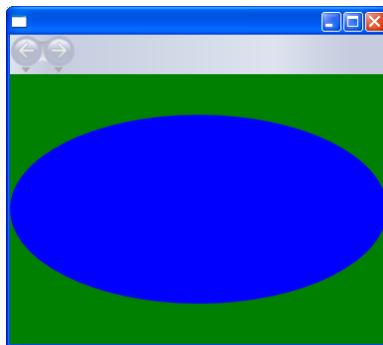


Figura 8-27 Fondo relleno con dos figuras geométricas

Note que para que la elipse no cubra toda el área vertical hemos indicado que su radio vertical (RadiusY) tiene valor 0.35.

Ahora cambiemos los colores verde y azul por un gradiente diagonal en el rectángulo y uno radial en la elipse (Listado 8-18 y Figura 8-28).

Listado 8-18 Figuras del DrawingBrush llenas con brochas de gradientes

```

<Page ... >
<Page.Background>
  <DrawingBrush Viewbox="0,0,1,1"
    ViewboxUnits="RelativeToBoundingBox"
    Viewport="0,0,1,1" ViewportUnits="RelativeToBoundingBox">
    <DrawingBrush.Drawing>
      <DrawingGroup>
        <GeometryDrawing>
          <GeometryDrawing.Brush>

```

```

<LinearGradientBrush StartPoint="0,0" EndPoint="0.35,0.51">
  <GradientStop Color="LightSteelBlue" Offset="0.1"/>
  <GradientStop Color="SteelBlue" Offset="0.5"/>
</LinearGradientBrush>
</GeometryDrawing.Brush>
<GeometryDrawing.Geometry>
  <RectangleGeometry Rect="0,0,1,1"/>
</GeometryDrawing.Geometry>
</GeometryDrawing>
<GeometryDrawing>
  <GeometryDrawing.Brush>
    <RadialGradientBrush>
      <GradientStop Color="#9fff" Offset="0"/>
      <GradientStop Color="#1fff" Offset="0.98"/>
    </RadialGradientBrush>
  </GeometryDrawing.Brush>
  <GeometryDrawing.Geometry>
    <EllipseGeometry Center="0.5,0.5" RadiusX="0.5" RadiusY="0.35"/>
  </GeometryDrawing.Geometry>
</GeometryDrawing>
</DrawingGroup>
</DrawingBrush.Drawing>
</DrawingBrush>
</Page.Background>
</Page>

```



Figura 8-28 Fondo relleno con figuras coloreadas con gradientes lineal y radial.

Divida imaginariamente el fondo de la Figura 8-28 en cuatro partes y observe el cuadrante de la esquina superior izquierda. Fíjese que esa imagen luce idéntica a la que rellena el fondo de la Figura 8-26. Recuerde que a través de la propiedad Viewbox se puede indicar cuál porción de toda la imagen es la que tomaremos para llenar. El Listado 8-19 finalmente define un fondo como el que buscábamos (Figura 8-29).

Listado 8-19 Fondo basado en figuras geométricas

```

<Page x:Class="BrushesAndPencilSamples.VideoBackground"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="VideoBackground">
<Page.Background>
    <DrawingBrush Viewbox="0.04,0,0.5,0.5"
ViewboxUnits="RelativeToBoundingBox"
        Viewport="0,0,1,1" ViewportUnits="RelativeToBoundingBox">
        <DrawingBrush.Drawing>
            <DrawingGroup>
                <GeometryDrawing>
                    <GeometryDrawing.Brush>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="0.35,0.51">
                            <GradientStop Color="LightSteelBlue" Offset="0.1"/>
                            <GradientStop Color="SteelBlue" Offset="0.5"/>
                        </LinearGradientBrush>
                    </GeometryDrawing.Brush>
                    <GeometryDrawing.Geometry>
                        <RectangleGeometry Rect="0,0,1,1"/>
                    </GeometryDrawing.Geometry>
                </GeometryDrawing>
                <GeometryDrawing>
                    <GeometryDrawing.Brush>
                        <RadialGradientBrush>
                            <GradientStop Color="#9fff" Offset="0"/>
                            <GradientStop Color="#1fff" Offset="0.98"/>
                        </RadialGradientBrush>
                    </GeometryDrawing.Brush>
                    <GeometryDrawing.Geometry>
                        <EllipseGeometry Center="0.5,0.5" RadiusX="0.5" RadiusY="0.35"/>
                    </GeometryDrawing.Geometry>
                </GeometryDrawing>
            </DrawingGroup>
        </DrawingBrush.Drawing>
    </DrawingBrush>
</Page.Background>
</Page>

```

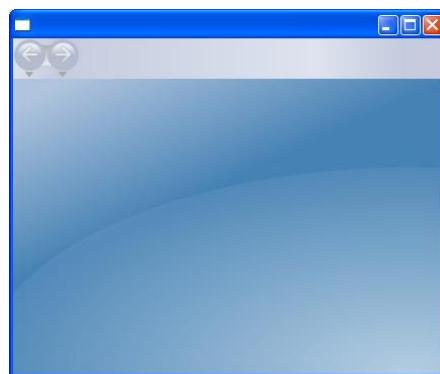


Figura 8-29 Ventana que se obtiene del Listado 8-22

Si observa detenidamente la Figura 8-26 notará que el panel que contiene los botones de reproducción del área de video también se ha rellenado con esta misma brocha. Si aún no lo cree, tome esta ventana y redúzcala a un tamaño similar (Figura 8-30) y convénzase de cómo se parecen. La figura se adapta gracias al valor por defecto Fill de la propiedad Stretch.



Figura 8-30 Ventana de la Figura 8-35 reducida verticalmente

Dibujos e imágenes

DrawingBrush no sólo es útil para formar rellenos con figuras sino que también puede contener imágenes y hasta vídeos. En esta sección veremos cómo crear rellenos formados por figuras e imágenes. La Figura 8-31 muestra un relleno combinado con estos dos tipos de dibujo (Listado 8-20). ¿Cómo lo ve? Si a usted le gusta decorar podrá haberse dado cuenta que con WPF se le abren las puertas para crear infinidad de tapices, hasta donde su imaginación le lleve.

Claro que con muchas de las herramientas profesionales de diseño que andan por ahí también se puedan lograr semejantes efectos. El chiste es que con WPF usted puede integrar esto a sus propias aplicaciones y relacionarlo con su propia lógica de negocio.



Figura 8-31 Relleno formado por figuras e imágenes

Listado 8-20 Combinación de diferentes brochas imágenes y figuras en un DrawingBrush

```
<Page x:Class="BrushesAndPencilSamples.DrawingBrush3"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="DrawingBrush3"
      Background="LightSteelBlue" >
<Grid>
  <Grid.Background>
    <DrawingBrush      Viewport="0,0,60,60"      ViewportUnits="Absolute">
```

```

<TileMode="FlipXY">
  <DrawingBrush.Drawing>
    <DrawingGroup>
      <ImageDrawing ImageSource="Images/Carita.bmp" Rect="0,0,30,30"/>
      <GeometryDrawing>
        <GeometryDrawing.Brush>
          <RadialGradientBrush>
            <GradientStop Color="#dfff" Offset="0"/>
            <GradientStop Color="#1000" Offset="0.75"/>
            <GradientStop Color="#1fff" Offset="0.7"/>
          </RadialGradientBrush>
        </GeometryDrawing.Brush>
        <GeometryDrawing.Geometry>
          <EllipseGeometry RadiusX="15" RadiusY="15" Center="45,15"/>
        </GeometryDrawing.Geometry>
      </GeometryDrawing>
      <GeometryDrawing>
        <GeometryDrawing.Brush>
          <LinearGradientBrush EndPoint="1,1">
            <GradientStop Color="#8fff" Offset="0"/>
            <GradientStop Color="#2000" Offset="1"/>
          </LinearGradientBrush>
        </GeometryDrawing.Brush>
        <GeometryDrawing.Geometry>
          <RectangleGeometry Rect="0,30,30,30"/>
        </GeometryDrawing.Geometry>
      </GeometryDrawing>
      <GeometryDrawing>
        <GeometryDrawing.Geometry>
          <RectangleGeometry Rect="30,30,30,30" RadiusX="4" RadiusY="4"/>
        </GeometryDrawing.Geometry>
        <GeometryDrawing.Brush>
          <ImageBrush ImageSource="Images/Carita.bmp"
                     Viewbox="95,95,150,150" ViewboxUnits="Absolute"/>
        </GeometryDrawing.Brush>
      </GeometryDrawing>
    </DrawingGroup>
  </DrawingBrush.Drawing>
</Grid.Background>
</Grid>
</Page>

```

El ejemplo del Listado 8-20 está formado por 4 elementos de dibujo distribuidos en 4 divisiones del Viewport, el elemento de la esquina superior izquierda es un ImageDrawing que se emplea para agregar al DrawingBrush una imagen ubicada en el rectángulo que se indique. Este elemento facilita la definición de imágenes dentro de un DrawingBrush, pero no ofrece toda la

versatilidad del ImageBrush, para lograrlo usted puede volver a emplear DrawingBrush o ImageBrush en el relleno de alguna figura por la que esté compuesto el DrawingBrush, como ocurre con las siguientes tres figuras:

Los otros tres elementos son figuras geométricas: En la esquina superior derecha hemos puesto una elipse que se ha llenado con un gradiente radial, en la inferior izquierda hemos puesto un rectángulo lleno con un gradiente diagonal, y finalmente en la esquina inferior derecha hemos puesto un rectángulo lleno con un ImageBrush al cual hemos especificado un Viewbox concentrado en una región de la imagen.

8.3.3 VisualBrush

Como hemos visto hasta aquí con ImageBrush podemos tener brochas cuyo contenido es una imagen, con DrawingBrush podemos llenar con figuras geométricas e imágenes. Esta es la tercera de las brochas herederas de TileBrush es VisualBrush, su contenido está formado por la representación visual de elementos de interacción como botones, cajas de texto, etiquetas, bordes, figuras, paneles, etc. que indica a través de la propiedad Visual de VisualBrush. En el siguiente ejemplo veremos como usando VisualBrush podemos "pequeños retratos" de partes de una interfaz de usuario.

Como seguro habrá experimentado, en algunas aplicaciones se suele emplear una miniatura (*Thumbnail*) o imagen reducida de una interfaz de usuario cuando por ejemplo esta última no cabe completamente en el área que disponemos para desplegarla. La Figura 8-32 muestra cómo Visual Studio presenta la miniatura del área de diseño de un esquema xsd, note que la miniatura a la derecha nos muestra todos los elementos que no caben en el área

Considere, por ejemplo, el código del Listado 8-21 que muestra el contenido de un panel como el UniformGrid y que nos produce la Figura 8-33.

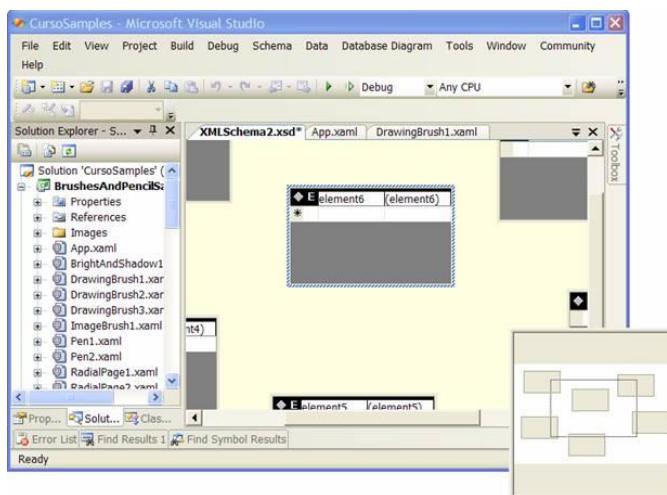


Figura 8-32 Miniatura de los elementos del área de diseño de esquemas xsd en Visual Studio

Listado 8-21 Página que muestra un formulario de datos personales

```
<Page x:Class="BrushesAndPencilSamples.VisualBrush1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="VisualBrush1" >
    <ScrollViewer Grid.Column="1">
        <UniformGrid Columns="2" Width="300" Margin="10,5"
            HorizontalAlignment="Left" VerticalAlignment="Top">
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16">Nombre:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16">Apellidos:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16">Título:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock VerticalAlignment="Center" FontSize="16">Edad:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16">Teléfono:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock VerticalAlignment="Center" FontSize="16">Email:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16">Dirección:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16">Profesión:</TextBlock>
            <TextBox Height="23" Width="100" Margin="5,10"/>
        </UniformGrid>
    </ScrollViewer>
</Page>
```

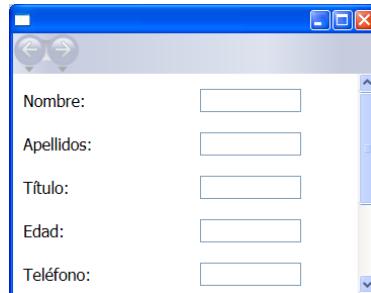


Figura 8-33 Formulario de datos personales

Note que en la Figura 8-33 no caben en la ventana todos los elementos del formulario y por ello se ha desplegado una barra de desplazamiento.

Modifiquemos un poco este código XAML (Listado 8-22) para dejar espacio en el cual mostrar una imagen reducida de dicha interfaz (Figura 8-34).

Listado 8-22 Formulario de datos personales con un borde a la izquierda que representa a la diapositiva

```
<Page ...>
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Border Width="180" Height="200" VerticalAlignment="Top"
        BorderBrush="RoyalBlue" BorderThickness="1" Margin="10,5">
        <TextBlock VerticalAlignment="Center"
            HorizontalAlignment="Center">
            Diapositiva
        </TextBlock>
    </Border>
    <ScrollViewer Grid.Column="1">
        <UniformGrid ...>
            ...
        </UniformGrid>
    </ScrollViewer>
</Grid>
</Page>
```

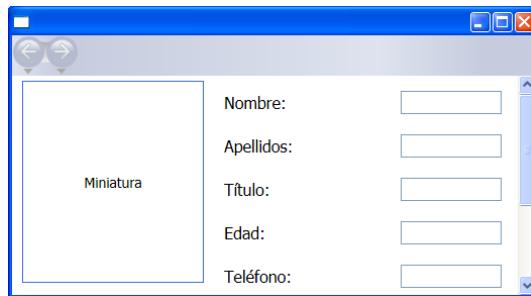


Figura 8-34 Formulario con espacio vacío a la izquierda para la miniatura de la interfaz

La Figura 8-34 muestra el lugar donde queremos poner la miniatura pero no lo que queremos ver en ella. Una miniatura del formulario debe mostrar todos los elementos que este contenga. Combinando apropiadamente la propiedad Visual de VisualBrush con las técnicas de enlace a datos (ver Lección **Enlace a Datos**) podemos llenar el área de la miniatura de la Figura 8-34 una foto imagen del UniformGrid (Listado 8-23 y Figura 8-35) y que como brocha lo que se rellena con ella podrá reducirse a voluntad.

Listado 8-23 VisualBrush cuyo Visual está enlazado con el elemento de nombre "content"

```

<Page ...>
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Border ...>
    <Border.Background>
      <VisualBrush Viewport="0.05,0.05,0.9,0.9"
                    Visual="{Binding
ElementName=formularioDePersona}"/>
    </Border.Background>
  </Border>
  <ScrollViewer Grid.Column="1">
    <UniformGrid ... Name="formularioDePersonaPLANILLA-D">
      ...
    </UniformGrid>
  </ScrollViewer>
</Grid>
</Page>

```

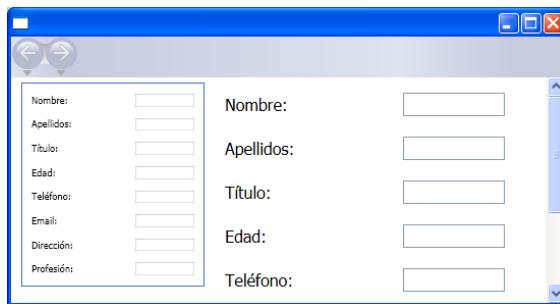


Figura 8-35 Miniatura que muestra una "foto" reducida de todo el formulario

Lo interesante a destacar en el Listado 8-23 es que con el enlace `Visual="{Binding ElementName=formularioDePersona}"` logramos decirle a la brocha que pinte con lo que esté visualizándose en el elemento de nombre `formularioDePersona`.

En otras lecciones usted podrá continuar usando y aplicando todas estas brochas a diferentes figuras que combinadas con las transformaciones y efectos de imágenes de WPF le permitirán lograr efectos todavía más atractivos como el efecto de reflejo que nos muestra la Figura 8-36.



Figura 8-36 Efecto reflejo donde se usa un VisualBrush y otros elementos de WPF que se verán en las próximas lecciones

8.4 Pinceles

Los pinceles, al igual que las brochas son recursos de dibujo que se emplean en el dibujo de figuras. Si las brochas se emplean para llenar las figuras los pinceles se emplean para dibujar el contorno de las figuras. Sin decirlo hemos estado empleando pinceles cada vez que hemos indicado el grosor de un borde o contorno (por ejemplo con las propiedades BorderThickness o StrokeThickness) y también cada vez que ha indicado la brocha con la que se dibuja un borde o contorno (BorderBrush o Stroke), que realmente indican la brocha con la cual se construye el pincel que dibuja el contorno

A diferencia de las brochas los pinceles no se especifican a través de una única propiedad sino mediante la combinación de varias propiedades que están presentes en casi todos los elementos de interacción. A continuación describiremos la mayoría de las propiedades que componen al pincel:

Brocha (BorderBrush o Stroke): Efectivamente, los pinceles están compuestos también con una brocha que emplean para llenar el contorno que estén dibujando. Si usted emplea la brocha del Listado 8-19 en el dibujo del contorno del borde del grid del Listado 8-24 obtendrá una imagen como la de la Figura 8-37.

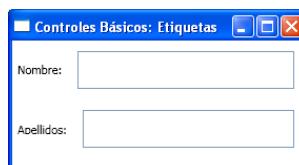


Figura 8-37 Contorno de un borde dibujado con un pincel basado en un DrawingBrush

Listado 8-24 Borde donde se ha indicado un DrawingBrush y un grosor para el contorno

```

<Page x:Class="BrushesAndPencilSamples.Pen1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Pen1" >
  <Grid>
    <Border Margin="15" BorderThickness="25" CornerRadius="10">
      <Border.BorderBrush>
        <DrawingBrush Viewbox="0.04,0,0.5,0.5"
                      ViewboxUnits="RelativeToBoundingBox"
                      Viewport="0,0,1,1" ViewportUnits="RelativeToBoundingBox">
          <DrawingBrush.Drawing>
            <DrawingGroup>
              <GeometryDrawing>
                <GeometryDrawing.Brush>
                  <LinearGradientBrush StartPoint="0,0" EndPoint="0.35,0.51">
                    <GradientStop Color="LightSteelBlue" Offset="0.1"/>
                    <GradientStop Color="SteelBlue" Offset="0.5"/>
                    <GradientStop Color="SteelBlue" Offset="1"/>
                  </LinearGradientBrush>
                </GeometryDrawing.Brush>
                <GeometryDrawing.Geometry>
                  <RectangleGeometry Rect="0,0,1,1"/>
                </GeometryDrawing.Geometry>
              </GeometryDrawing>
              <GeometryDrawing>
                <GeometryDrawing.Brush>
                  <RadialGradientBrush>
                    <GradientStop Color="#9fff" Offset="0"/>
                    <GradientStop Color="#1fff" Offset="0.98"/>
                  </RadialGradientBrush>
                </GeometryDrawing.Brush>
                <GeometryDrawing.Geometry>
                  <EllipseGeometry Center="0.5,0.5" RadiusX="0.5" RadiusY="0.35"/>
                </GeometryDrawing.Geometry>
              </GeometryDrawing>
            </DrawingGroup>
          </DrawingBrush.Drawing>
        </DrawingBrush>
      </Border.BorderBrush>
    </Border>
  </Grid>
</Page>

```

Grosor (BorderThickness o StrokeThickness): Para que se observara mejor en el Listado 8-24 se ha exagerado el grosor del borde al hacer BorderThickness = ""25"". Pero con el ...Thickness no sólo se indica el grosor de todo el contorno, también se pueden indicar diferentes grosores para los extremos izquierdo, derecho, superior e inferior del grosor: Si en vez de darle un solo valor

"25" al grosor le diéramos el par de valores ""6,2"" WPF traduce esto a que queremos darle un valor de 6 pixels a los laterales del contorno (izquierda y derecha) y un valor de 2 pixels a los laterales superior e inferior (Figura 8-38 y Listado 8-25).



Figura 8-38 Borde con grosores diferentes en los laterales horizontales y verticales.

Listado 8-25 BorderThickness con valor compuesto

```
<Page ...>
<Grid>
    <Border BorderThickness="6,2" ...>
        <Border.BorderBrush>
            ...
        </Border.BorderBrush>
    </Border>
</Grid>
</Page>
```

Como podrá haber inferido, realmente se puede indicar un valor diferente para cada lateral. El Listado 8-26 y la Figura 8-39 nos muestran cómo lograr el efecto de una lengüeta de un **TabControl** combinando los valores de **CornerRadius** y **BorderThickness**. Note que a la propiedad **BorderThickness** se le han dado los cuatro valores ""2,4,2,0"" donde el 0 se interpreta como que el lateral inferior no tiene ningún borde.



Figura 8-39 Esbozo de una lengüeta logrado por combinación de **CornerRadius** y **BorderThickness** en un **Border**

Listado 8-26 BorderThickness con grosor diferente en cada lateral

```
<Page ...>
<Grid>
    <Border Margin="15" BorderThickness="2,4,2,0"
CornerRadius="6,6,0,0">
```

```

<Border.BorderBrush>
...
</Border.BorderBrush>
<TextBlock VerticalAlignment="Center"
HorizontalAlignment="Center">
    Tab 1
</TextBlock>
</Border>
</Grid>
</Page>

```

Terminaciones de línea (StrokeStartLineCap y StrokeEndLineCap): Observe detenidamente los extremos de las líneas que componen la flecha de la Figura 8-40 descrita por el Listado 8-27. ¿No es cierto que se ven muy cuadrados? Con las propiedades StrokeStartLineCap y StrokeEndLineCap se puede determinar cómo se dibuja el inicio y el fin de un borde que corresponda a una figura abierta como la línea (Line) o la poligonal (Polyline). Estas propiedades pueden tomar valores como Round y Triangle que se ilustran en la Figura 8-41.

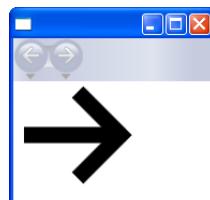


Figura 8-40 Flecha lograda con una línea y una poligonal

Listado 8-27 Línea y poligonal que describen una flecha

```

<Page x:Class="BrushesAndPencilsSamples.Pen2"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Pen2">
    <Grid>
        <Line X1="10" X2="90" Y1="50" Y2="50"
              StrokeThickness="15" Stroke="Black" />
        <Polyline Points="60,10 100,50 60,90"
                  StrokeThickness="15" Stroke="Black"/>
    </Grid>
</Page>

```



Figura 8-41 Efecto de los valores Triangle y Round en las propiedades StrokeStartLineCap y StrokeEndLineCap

Junturas (StrokeLineJoin): En la Figura 8-41 usted habrá notado que los extremos de las figuras se han redondeado con el valor Round de StrokeStartLineCap y StrokeEndLineCap sin embargo la punta de la flecha sigue siendo puntiaguda. Esto ocurre porque estas propiedades sólo afectan a los extremos de las figuras y la punta de la flecha es en este caso la unión de dos segmentos de un Polyline. StrokeLineJoin es la propiedad que permite indicar la forma en que se unen las diferentes líneas que forman una figura. El valor por defecto de esta propiedad es Miter. La Figura 8-42 nos muestra dos ejemplos de valores que puede tomar esta propiedad.



Figura 8-42 Efecto de los valores Bevel y Round de la propiedad StrokeLineJoin.

Salpicaduras (StrokeDashArray y StrokeDashCap): Los contornos pueden dibujarse en forma discontinua como Ud. puede hacer en MS Office. Usando la propiedad StrokeDashArray se puede definir la discontinuidad con la que se quiere que se dibuje un contorno. El valor que se le da a esta propiedad se interpreta como un array de valores enteros que define las longitudes continuas y el espacio vacío de la línea. Por ejemplo si se hace StrokeDashArray = "2 1", esto se interpreta como que la línea discontinua está formada por segmentos formados por dos (2) segmentos (del grosor y longitud correspondientes con el valor de StrokeThickness) continuos seguido de un (1) segmento del mismo grosor pero vacío (Figura 8-43 y Listado 8-28).

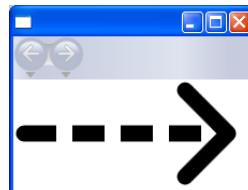


Figura 8-43 Línea de la flecha dibujada con un contorno discontinuo

Listado 8-28 Definición de líneas discontinuas

```

<Page ...>
<Grid>
  <Line X1="10" X2="190" Y1="50" Y2="50"
    StrokeStartLineCap="Round" StrokeEndLineCap="Round"
    StrokeDashArray="2 1"
    StrokeThickness="15" Stroke="Black" />
    <Polyline      Points="160,10      200,50      160,90"
      StrokeStartLineCap="Round"
      StrokeEndLineCap="Round" StrokeLineJoin="Round"
      StrokeThickness="15" Stroke="Black"/>
  </Grid>
</Page>
```

Si en vez de definirlo como "2 1" usted indica que el valor de `StrokeDashArray` sea "1 2 1", WPF dibuja 1 un segmento continuo, luego 2 vacíos, 1 continuo, 1 vacío, 2 continuos, 1 vacío, así sucesivamente hasta el final del contorno (Figura 8-44).

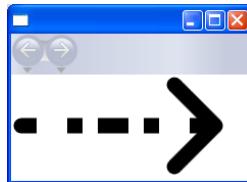


Figura 8-44 Línea discontinua definida con una cantidad impar de bloques

Al igual que las junturas y las terminaciones, las salpicaduras pueden dibujarse de forma no cuadrada. La propiedad `StrokeDashCap` permite indicar como se dibujan las terminaciones de cada segmento continuo para cada fragmento de las salpicaduras. La Figura 8-45 se logra haciendo esta propiedad a `StrokeDashCap` = "Round".

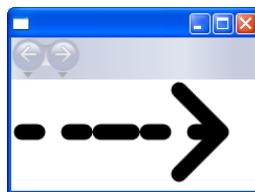


Figura 8-45 Líneas discontinuas redondeadas

8.4.1 Pinceles en las figuras geométricas

Las figuras geométricas que se usan para componer un `Path` o un `DrawingBrush` usan el concepto de pluma (`Pen`) para definir sus contornos. A través de una propiedad `Pen` donde se indican las características del pincel (o la pluma) a través de sus propiedades `Brush`, `Thickness`, `StartLineCap`, `EndLineCap`, `LineJoin`, `DashArray` y `DashCap`.

Lección 9 Efectos Visuales

En lecciones anteriores hemos visto cómo utilizar las figuras brochas y pinceles para mejorar la apariencia de una aplicación. En esta lección veremos algunos efectos que se podrán lograr para aumentar el realismo de los elementos visuales que usted utilice de WPF. Se verá en esta lección cómo lograr efectos de transparencia, sombra, resplandor, relieve y desenfoque.

9.1 Efectos de transparencia

En la lección de **Brochas** usted ha tenido un primer encuentro con la transparencia y semitransparencia empleando la componente Alfa de transparencia en los colores. Recordemos

que si usted utiliza el color #FFFF estará refiriéndose al color Blanco, sin embargo el color #0FFF es un color completamente transparente por tener su componente Alfa con valor 0. Por otro lado valores intermedios como #8FFF son variaciones de transparencia del color blanco haciendo que las figuras que se rellenen con un color como éste den un efecto de aclarar o iluminar el fondo sobre el que se ponen.

Aunque la componente Alfa de los colores suele ser suficiente para lograr muchos de los efectos de dibujo en WPF, hay otros efectos de transparencia que pueden ser más complicados de lograr empleando este recurso. Todos los elementos de WPF que son interfaz de usuario cuentan con dos propiedades con las que se pueden lograr otros efectos de transparencia: Opacity y OpacityMask.

9.1.1 Opacidad

La opacidad es el efecto contrario a la transparencia. En WPF la opacidad se manifiesta a través de la propiedad Opacity definida en UIElement. Esta propiedad puede tomar valores reales entre 0 y 1. Cero indica que el elemento es completamente transparente (no se ve) y uno, que es valor por defecto, indica que el elemento se muestra exactamente tal y como fue coloreado. Aclaremos que si un elemento ha sido coloreado con un color transparente o semitransparente, la opacidad 1 respeta este carácter semitransparente. El valor que se da a esta opacidad aumenta o disminuye de manera uniforme el grado de transparencia del elemento completo. En el Listado 9-1 hemos puesto un botón semitransparente (con opacidad 0.5) sobre un fondo cuadriculado (Figura 9-1) que nos permite distinguir el grado de transparencia del segundo botón. Si usted llega a distinguir las cuadriculas tras el botón es producto del grado de opacidad de la propiedad Opacity que se le ha dado al botón Cancelar. El valor predeterminado de Opacity es 1.0 que significa opacidad total, es decir no deja ver nada de lo que esté detrás.

Listado 9-1 Página con dos botones: uno opaco y el otro semitransparente

```
Page x:Class="VisualEffects.Opacity"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Opacity">
<Canvas>
    <Canvas.Background>
        <DrawingBrush Viewport="0,0,10,10" ViewportUnits="Absolute"
            Viewbox="0,0,10,10" ViewboxUnits="Absolute"
            TileMode="Tile">
            <DrawingBrush.Drawing>
                <GeometryDrawing Brush="White">
                    <GeometryDrawing.Pen>
                        <Pen Brush="LightSteelBlue" Thickness="1"/>
```

```

</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <RectangleGeometry Rect="0,0,10,10"/>
</GeometryDrawing.Geometry>
</GeometryDrawing>
</DrawingBrush.Drawing>
</DrawingBrush>
</Canvas.Background>
<Button Margin="10,10" Width="80" Height="30">
  Aceptar
</Button>
<Button Margin="120,10" Opacity="0.5" Width="80" Height="30">
  Cancelar
</Button>
</Canvas>
</Page>

```

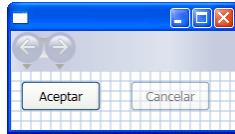


Figura 9-1 Dos botones. El segundo es semitransparente

Observe en la Figura 9-1 que en el botón *Cancelar* están semitransparente su fondo, su borde y su texto. El efecto de la propiedad Opacity se aplica al elemento y a todos los elementos que éste contenga. Si por ejemplo hacemos que la página sea color tomate obtendríamos una ventana idéntica a la de la Figura 9-1 pero si pusiéramos la opacidad del Canvas a un 70% (Listado 9-2) usted podrá observar en la Figura 9-2 que todo el canvas toma un color rojizo al igual que los botones por la semitransparencia obtenida a través de la opacidad del Canvas, y si observa detenidamente notará también como el segundo botón se hace ligeramente menos visible por la misma razón.

Listado 9-2 Ejemplo anterior con menos opacidad en el Canvas sobre un fondo color tomate

```

Page x:Class="VisualEffects.Opacity"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Opacity" Background="Tomato">
<Canvas Opacity="0.7">
  <Button ... />
  <Button ... />
</Canvas>
</Page>

```

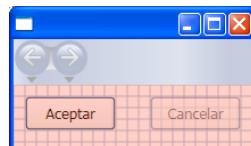


Figura 9-2 Canvas semitransparente sobre una página color tomate

9.1.2 Máscara de opacidad

En la Lección **Brochas** hemos mostrado un Border con una brocha que rellena su fondo con un gradiente lineal semitransparente que nos da un efecto cristalino como el de la Figura 9-3.



Figura 9-3 Botón con efecto cristalino

Para obtener un efecto como el de la Figura 9-3 no es utilizable la propiedad Opacity porque, como hemos dicho con anterioridad, esta propiedad aplica la transparencia uniformemente y si observa detenidamente en la figura barriendo con la vista de abajo hacia arriba notará como la transparencia en el fondo del Border se va perdiendo de modo que la parte inferior se ve más transparente que la superior.

Con un poco de esfuerzo es posible obtener un borde como el de la Figura 9-3 ajustando convenientemente el componente Alfa de los colores de un gradiente vertical. Le invitamos a que lo intente tomando GoldenRod como color base del gradiente.

Las imágenes con efectos de transparencia como la de la Figura 9-3 tienen algunos elementos que dificultan su definición en XAML a partir de la componente Alfa de los colores:

1. Conocer que se quiere lograr un efecto a partir de un color particular, como GoldenRod en nuestro ejemplo, fuerza a tener que conocer cuáles son los valores de saturación de rojo, verde y azul del color para añadirle luego el componente Alfa en la propiedad Color de cada GradientStop del gradiente vertical con que se rellena la figura.
2. En ocasiones el color inicial no es conocido durante la edición del XAML sino que se puede querer obtener a partir de enlace con otros datos (Bindings) o asignado desde la lógica mediante código C# durante la ejecución de la aplicación.
3. Si observa detenidamente de nuevo la Figura 9-3 notará que en la parte superior se dejar ver un efecto de brillo. Hemos aprendido en lecciones

anteriores a lograr este efecto añadiendo figuras en la parte superior rellenas con gradientes que utilizan blancos semitransparentes. Para evitar que el brillo sea demasiado intenso usted tendría que probar una y otra vez con diferentes valores del componente Alfa en estos blancos del gradiente hasta alcanzar la imagen deseada.

Estas limitantes pudieran ser superadas si la transparencia del fondo pudiera definirse de manera separada a la del color de relleno y de los demás efectos que se quieran lograr. WPF nos brinda la propiedad OpacityMask que tiene un efecto parecido al de Opacity con la ventaja adicional de que con OpacityMask podemos definir también gradientes de opacidad. Veremos a continuación cómo se utiliza esta propiedad para alcanzar el efecto de la Figura 9-3 sin mayores dificultades.

Construyamos primero la imagen de la Figura 9-3 con el color GoldenRod de fondo y el efecto de brillo pero sin transparencia (Listado 9-3 y Figura 9-4).

Listado 9-3 Border con efecto de brillo

```
<Page x:Class="VisualEffects.OpacityMask"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="OpacityMask">
  <Canvas>
    <Canvas.Background>
      <DrawingBrush Viewport="0,0,10,10" ViewportUnits="Absolute"
                    Viewbox="0,0,10,10" ViewboxUnits="Absolute"
                    TileMode="Tile">
        <DrawingBrush.Drawing>
          <GeometryDrawing Brush="White">
            <GeometryDrawing.Pen>
              <Pen Brush="LightSteelBlue" Thickness="1"/>
            </GeometryDrawing.Pen>
            <GeometryDrawing.Geometry>
              <RectangleGeometry Rect="0,0,10,10"/>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
        </DrawingBrush.Drawing>
      </DrawingBrush>
    </Canvas.Background>
    <Border CornerRadius="10" Width="120" Height="65"
          BorderBrush="DarkGoldenrod" BorderThickness="2"
          Canvas.Left="10" Canvas.Top="10">
      <Border CornerRadius="6" Background="Goldenrod" >
        <Border CornerRadius="6" Height="12" VerticalAlignment="Top">
```

```

<Border.Background>
  <LinearGradientBrush EndPoint="0,1">
    <GradientStop Color="#dFFF" Offset="0"/>
    <GradientStop Color="Transparent" Offset="1"/>
  </LinearGradientBrush>
</Border.Background>
</Border>
</Border>
</Canvas>
</Page>

```

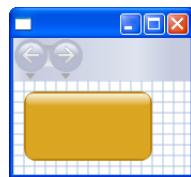


Figura 9-4 Border de la Figura 9-3 sin transparencia

Seguidamente aplicaremos la propiedad OpacityMask al segundo Border del Listado 9-3 para que no se afecte la opacidad del contorno de la figura (Listado 9-4). La propiedad OpacityMask es de tipo Brush y a ella podemos asociarle cualquier tipo de brocha de WPF. OpacityMask toma la brocha que se le indique y "recolorea" el elemento cambiando cada componente Alfa del fondo del elemento con el componente Alfa de los colores que componen la brocha. Observe en el Listado 9-4 que hemos definido un gradiente lineal vertical para la propiedad basado en negros semitransparentes (#C000 y #4000). El resultado es justamente la Figura 9-3 donde no se ve obscurecimiento en el fondo del Border porque OpacityMask sólo ha hecho caso de los valores **C** y **4** que representan la componente Alfa de los colores #C000 y #4000 con que se definió el gradiente y acorde con esto ha modificado la opacidad del relleno del Border y su contenido de modo que luzca como esperábamos.

Listado 9-4 Máscara de opacidad aplicada a un borde
<pre> <Page ...> <Canvas> ... <Border CornerRadius="10" Width="120" Height="65" BorderBrush="DarkGoldenrod" BorderThickness="2" Canvas.Left="10" Canvas.Top="10"> <Border CornerRadius="6" Background="Goldenrod" > <Border CornerRadius="6" Height="12" VerticalAlignment="Top"> <Border.Background> <LinearGradientBrush EndPoint="0,1"> </pre>

```

<GradientStop Color="#dFFF" Offset="0"/>
<GradientStop Color="Transparent" Offset="1"/>
</LinearGradientBrush>
</Border.Background>
</Border>
<Border.OpacityMask>
<LinearGradientBrush EndPoint="0,1">
<GradientStop Color="#C000" Offset="0"/>
<GradientStop Color="#4000" Offset="1"/>
</LinearGradientBrush>
</Border.OpacityMask>
</Border>
</Border>
</Canvas>
</Page>

```

Pruebe usted mismo a utilizar RadialGradientBrush o DrawingBrush en vez de LinearGradientBrush y disfrute de los efectos.

En la Lección **Brochas** se culmina el estudio de VisualBrush mostrando varios controles con un efecto de reflejo como el de la Figura 9-5.



Figura 9-5 Efecto reflejo donde se usa VisualBrush, OpacityMask y otros elementos de WPF que se verán en las próximas lecciones

Aunque en esta lección aún no estamos listos para descubrir todos los trucos WPF del efecto reflejo, le adelantaremos que OpacityMask tiene mucho que ver para obtener estos (Listado 9-5 y Figura 9-6).

Listado 9-5 Borde y Botón con máscara de opacidad

```

<Page ...>
<Canvas>
<Border ...>
...

```

```

</Border>
<Button Margin="145,10" FontSize="30" Width="140" Height="65">
    Insertar
    <Button.OpacityMask>
        <LinearGradientBrush EndPoint="0,1">
            <GradientStop Color="#A000" Offset="0"/>
            <GradientStop Color="#0000" Offset="1"/>
        </LinearGradientBrush>
    </Button.OpacityMask>
</Button>
</Canvas>
</Page>

```

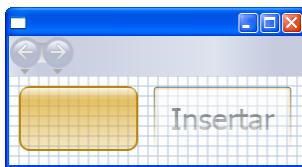


Figura 9-6 Borde y Botón con efecto de máscara de opacidad

9.2 Efectos de Bitmap

Los efectos de Bitmap o BitmapEffects no son efectos que se aplican a imágenes .bmp que usted ponga en su aplicación, como aparentemente le pudiera sugerir el nombre de esta sección. BitmapEffect es a la vez el nombre de una propiedad de UIElement, que representa a un conjunto de efectos que usted puede aplicar a todos los elementos WPF. Al final de esta sección explicaremos cuál es el origen del nombre BitmapEffect y la repercusión que tiene en el funcionamiento de su aplicación, pero antes veremos de cuáles efectos estamos hablando:

9.2.1 Efecto de Sombra

El efecto de sombra es un efecto que permite dar realismo a las aplicaciones. Tanto la iluminación como la sombra son efectos presentes en el mundo real que nos rodea. El efecto de sombra le da cuerpo tridimensional a los elementos visuales, aportando sensación de profundidad al carácter plano de los textos, figuras e imágenes que componen la apariencia de la aplicación. WPF ofrece la posibilidad de aplicar sombra a todos los elementos de la interfaz de usuario a través del tipo DropShadowBitmapEffect. Observe el carácter plano que tiene el texto (por cierto ya mejorado por el gradiente en el fondo de la ventana que se ha rellenado con una brocha diagonal) de la Figura 9-7 (Listado 9-6).



Figura 9-7 Texto plano en un Grid relleno con una brocha diagonal

Listado 9-6 Grid relleno con un gradiente diagonal con un bloque de texto blanco

```
<Page x:Class="VisualEffects.DropShadow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="DropShadow">
    <Grid>
        <Grid.Background>
            <LinearGradientBrush EndPoint="1,1">
                <GradientStop Color="RoyalBlue" Offset="0"/>
                <GradientStop Color="AliceBlue" Offset="1"/>
            </LinearGradientBrush>
        </Grid.Background>
        <TextBlock Foreground="White"   FontSize="24"   Margin="20"
                  FontFamily="Calibri"           VerticalAlignment="Top"
                  HorizontalAlignment="Left">
            Autenticación
        </TextBlock>
    </Grid>
</Page>
```

Compare la imagen de la Figura 9-7 con la de la Figura 9-8 y observe como el texto se muestra con mejor nitidez y realismo al aplicar el efecto de sombra que le hemos añadido al TextBlock en el Listado 9-7.



Figura 9-8 Texto de la Figura 9-7 con efecto de sombra

Listado 9-7 Texto con efecto de sombra

```
<Page ...>
    <Grid>
        <Grid.Background>
            ...
        </Grid.Background>
    </Grid>
</Page>
```

```

</Grid.Background>
<TextBlock ...>
    Autenticación
    <TextBlock.BitmapEffect>
        <DropShadowBitmapEffect Color="Black"
            Opacity="0.8" ShadowDepth="1.4" Direction="315"
            Softness="0.2"/>
    </TextBlock.BitmapEffect>
</TextBlock>
</Grid>
</Page>

```

Este efecto de sombra tiene un conjunto de propiedades que le permiten ajustar la sombra a su gusto. La propiedad Color permite indicar el color con que se dibuja la sombra. Normalmente se emplean colores oscuros para la sombra. Las propiedades que más realismo dan a la brocha son ShadowDepth, Softness, Opacity y Direction.

La profundidad del elemento respecto del fondo se logra con la combinación propiedad ShadowDepth. Esta propiedad indica la separación de la brocha respecto del elemento dado en pixels. A mayor valor de esta propiedad más lejos estará la sombra del elemento al que se aplica (Figura 9-9).

Softness regula la suavidad de los bordes de la brocha y sugiere el grado de refracción que tiene el objeto que provocó la sombra. Por regla general los objetos más opacos producen bordes menos suaves y los cristalinos proyectan una sombra más suave. Este efecto de suavidad en los bordes varía también con la distancia del elemento respecto del fondo donde se proyecta la sombra. A mayor valor de esta propiedad mayor suavidad usted obtendrá.

Por su parte Opacity indica el grado de transparencia de la sombra respecto del fondo. Su valor sugiere el grado de intensidad que se supone tenga la luz que ha proyectado la sombra, a mayor intensidad, más opacidad.

Finalmente Direction es la propiedad que permite indicar el ángulo (en grados) de inclinación de la luz con respecto al fondo.

En ocasiones resulta complicado determinar cuáles son los valores apropiados que debemos darle a estas propiedades. Nuestra sugerencia es que juegue a la sombra china una vez más y observe cómo se comporta la sombra cuando usted acerca o aleja sus manos de la pared prestando especial atención a la opacidad y suavidad en los bordes. Con el código del Listado 9-8 hemos alejado un poco el texto de su fondo (dándole valor 13 a la propiedad ShadowDepth) para dar mayor sensación de profundidad (Figura 9-9) .



Figura 9-9 Efecto de sombra distante

Listado 9-8 Variación de la sombra para dar efecto de mayor profundidad

```
<Page ...>
<Grid>
    ...
    <TextBlock ...>
        Autenticación
        <TextBlock.BitmapEffect>
            <DropShadowBitmapEffect Color="Black" Opacity="0.4"
                ShadowDepth="12" Direction="315" Softness="0.3"/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</Grid>
</Page>
```

Además de las propiedades anteriores DropShadowBitmapEffect tiene una propiedad Noise que puede utilizarse para simular "rugosidad" en el fondo, algo así como que la sombra estuviera proyectándose en una superficie más rugosa (como una pared). Para esta propiedad se suele emplear un color oscuro parecido al fondo. En la Figura 9-10 mostramos el empleo de Noise con valor 0.2 empleando azul oscuro en el color de la sombra.

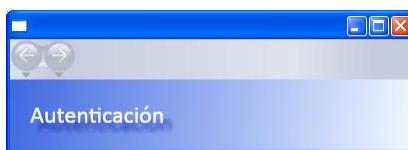


Figura 9-10 Sombra con distorsión

9.2.2 Efecto de resplandor

El resplandor (OuterGlowBitmapEffect) es otro efecto de luz muy llamativo que se utiliza para resaltar un elemento entre varios dentro de la aplicación. En el código del Listado 9-9 hemos aplicado este efecto para resaltar a un botón de entre los demás como puede apreciarse en la Figura 9-11.

Listado 9-9 Efecto de resplandor aplicado a un botón

```

<Page x:Class="VisualEffects.OuterGlow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="OuterGlow" >
<Page.Background>
<LinearGradientBrush EndPoint="1,1">
<GradientStop Color="RoyalBlue" Offset="0"/>
<GradientStop Color="AliceBlue" Offset="1"/>
</LinearGradientBrush>
</Page.Background>
<StackPanel Orientation="Horizontal">
<Button Margin="10" Padding="5,2">
    Nuevo
    <Button.BitmapEffect>
        <OuterGlowBitmapEffect     GlowColor="White"     GlowSize="8"
        Opacity="0.8"/>
    </Button.BitmapEffect>
</Button>
<Button Margin="10" Padding="5,2">Abrir</Button>
<Button Margin="10" Padding="5,2">Guardar</Button>
<Button Margin="10" Padding="5,2">Cerrar</Button>
</StackPanel>
</Page>

```



Figura 9-11 El primer botón tiene aplicado un efecto de resplandor

Este efecto crea una aureola alrededor del elemento del tamaño que se indique en la propiedad GlowSize con el color de GlowColor y el grado de opacidad asignado a la propiedad Opacity. Para este efecto también está disponible la propiedad Noise como en el efecto de sombra.

El efecto de resplandor exterior es también muy útil para simular sombras uniformes y efectos de enmarcado de algunas figuras utilizando colores más bien oscuros (Listado 9-10 y Figura 9-12).

Listado 9-10 Efectos de resplandor aplicados a un Border y un botón

```

<Page ... Background="RoyalBlue">
<Border Margin="8" CornerRadius="10" Background="White">
<StackPanel Orientation="Horizontal">

```

```

<Button Margin="10" Padding="5,2">
    Nuevo
    <Button.BitmapEffect>
        <OuterGlowBitmapEffect     GlowColor="Yellow"     GlowSize="8"
        Opacity="0.6"/>
    </Button.BitmapEffect>
</Button>
<Button Margin="10" Padding="5,2">Abrir</Button>
<Button Margin="10" Padding="5,2">Guardar</Button>
<Button Margin="10" Padding="5,2">Cerrar</Button>
</StackPanel>
<Border.BitmapEffect>
    <OuterGlowBitmapEffect     GlowColor="Black"     Opacity="0.5"
    GlowSize="6"/>
</Border.BitmapEffect>
</Border>
</Page>

```



Figura 9-12 Efecto de resplandor obscuro aplicado a un Border

Si aún no nota el efecto compárelo con la Figura 9-13 que no tiene aplicado el efecto de resplandor obscuro.

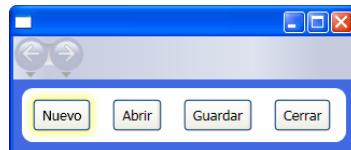


Figura 9-13 Imagen de la Figura 9-12 sin efecto de resplandor obscuro

9.2.3 Efecto de desenfoque

El desenfoque es un efecto visual que usted puede apreciar en el mundo real. El más común es el efecto de desenfoque por lejanía.

Imagine que usted se aleja considerablemente de un objeto que originalmente tenía muy cerca. A medida que aumenta la distancia, muchos detalles del objeto se van haciendo imperceptibles y la imagen que originalmente teníamos del objeto no sólo se reduce de tamaño sino que también pierde calidad de sus detalles. Tomemos una ventana de autenticación como la de la Figura 9-14 y Listado 9-11.



Figura 9-14 Ventana de autenticación

Listado 9-11 Página de autenticación

```
<Page x:Class="VisualEffects.Blur"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Blur" >
<Grid>
    <Grid.Background>
        <LinearGradientBrush EndPoint="0,0.5" SpreadMethod="Reflect">
            <GradientStop Color="DarkBlue" Offset="0"/>
            <GradientStop Color="RoyalBlue" Offset="1"/>
        </LinearGradientBrush>
    </Grid.Background>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <TextBlock     Foreground="White"      FontSize="24"      Margin="20"
FontFamily="Calibri"
        VerticalAlignment="Center" HorizontalAlignment="Left">
        Autenticación
    <TextBlock.BitmapEffect>
        <DropShadowBitmapEffect Color="Black" Opacity="0.8"
            ShadowDepth="1.4" Direction="315" Softness="0.2"/>
    </TextBlock.BitmapEffect>
    </TextBlock>
    <TextBlock Margin="30,15" Grid.Row="1" FontSize="14" Foreground="White"
        VerticalAlignment="Center">
        Nombre de usuario:
    <TextBlock.BitmapEffect>
        <DropShadowBitmapEffect Color="Black" Opacity="0.8"
            ShadowDepth="1.4" Direction="315" Softness="0.2"/>
```

```

</TextBlock.BitmapEffect>
</TextBlock>
<TextBox Margin="170,5,20,5" Height="25" Grid.Row="1" Padding="2,1"
    VerticalAlignment="Center"/>
<TextBlock Margin="30,15" Grid.Row="2" FontSize="14" Foreground="White"
    VerticalAlignment="Center">
Contraseña:
<TextBlock.BitmapEffect>
<DropShadowBitmapEffect Color="Black" Opacity="0.8"
    ShadowDepth="1.4" Direction="315" Softness="0.2"/>
</TextBlock.BitmapEffect>
</TextBlock>
<PasswordBox Margin="170,5,20,5" Height="25" Grid.Row="2" Padding="2,1"
    VerticalAlignment="Center"/>
<Button Padding="10,4" Margin="20,10" Grid.Row="3"
    VerticalAlignment="Center" HorizontalAlignment="Right" >
    Entrar
</Button>
</Grid>
</Page>

```

Supongamos que se ha intentado una autenticación incorrecta y se quiere alertar sobre ello con un mensaje. Lo usual es que el mensaje aparezca al frente de la ventana y separado de ella, de modo de dar la sensación que se sobrepone a ésta.

Para ello hacemos que el mensaje aparezca en un bloque de texto dentro de un Border que además contenga un botón. Como se ilustra en la Figura 9-15 y Listado 9-12.



Figura 9-15 Ventana de autenticación con mensaje de alerta

Listado 9-12 Border puesto encima de la ventana que representa el mensaje de alerta

```

<Page ...>
<Grid>
...

```

```

<TextBlock .../>
<TextBlock .../>
<TextBox .../>
<TextBlock .../>
<PasswordBox .../>
<Button .../>
<Border CornerRadius="6,6,4,4" BorderBrush="Gray" BorderThickness="1"
        Grid.RowSpan="4"           HorizontalAlignment="Center"
VerticalAlignment="Center">
    <Border.Background>
        <LinearGradientBrush EndPoint="0,1">
            <GradientStop Color="White" Offset="0.4"/>
            <GradientStop Color="LightGray" Offset="1"/>
        </LinearGradientBrush>
    </Border.Background>
    <StackPanel>
        <TextBlock Margin="10,15,10,10">
            Nombre de usuario y contraseña incorrectos
        </TextBlock>
        <Button Margin="10" HorizontalAlignment="Center"
VerticalAlignment="Center">
            Aceptar
        </Button>
    </StackPanel>
</Border>
</Grid>
</Page>

```

En el Listado 9-12 hemos puesto directamente el Border encima de los demás elementos en la ventana. En un escenario real este Border aparecería como resultado de un evento y una verificación de la contraseña que aquí nos hemos ahorrado para centrarnos en el objetivo principal de esta sección que es el efecto de desenfoque.

Observe que en la Figura 9-14 el mensaje luce muy plano, al mismo nivel de profundidad del resto de los controles. Una primera aproximación para dar efecto de distancia es aplicar un efecto de sombra al Border como ya hemos visto anteriormente (Figura 9-16 y Listado 9-13).



Figura 9-16 Mensaje con efecto de sombra para simular distancia

Listado 9-13 Mensaje con efecto de sombra

```
<Page ...>
<Grid>
    <Grid>
        ...
        <TextBlock .../>
        <TextBlock .../>
        <TextBox .../>
        <TextBlock .../>
        <PasswordBox .../>
        <Button .../>
    </Grid>
    <Border ...>
        ...
        <Border.BitmapEffect>
            <DropShadowBitmapEffect Color="Black" Opacity="0.4"
                Softness="0.8" Direction="315"
                ShadowDepth="16" />
        </Border.BitmapEffect>
    </Border>
</Grid>
</Page>
```

Observe que en el Listado 9-13 hemos agregado un Grid de modo que los controles de la ventana de autenticación queden aislados en otro Grid. En el Listado 9-14 hemos aplicado efecto de desenfoque al Grid que contiene los controles originales para dar con ello la sensación de distancia del mensaje respecto al fondo. Compruébelo comparando la Figura 9-16 con la Figura 9-17.

Listado 9-14 Efecto de desenfoque aplicado al fondo

```
<Page ...>
```

```

<Grid>
    <Grid>
        ...
        <Grid.BitmapEffect>
            <BlurBitmapEffect Radius="1.5"/>
        </Grid.BitmapEffect>
    </Grid>
    <Border ...>
        ...
    </Border>
</Grid>
</Page>

```

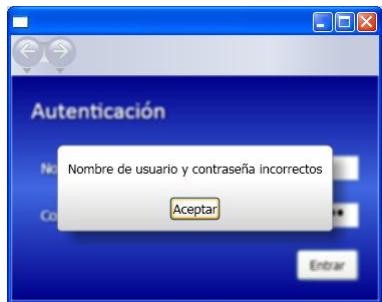


Figura 9-17 Efecto de desenfoque aplicado a un fondo para simular distancia

La propiedad Radius regula el grado de desenfoque que usted puede alcanzar. A mayor valor de esta propiedad más desenfoque se obtendrá.

A continuación se describen otros fenómenos que pueden provocar un desenfoque. Le invitamos a que los desarrolle usted mismo:

Enfoque/Desenfoque: Usted puede notar este efecto visual ahora mismo si hace el siguiente experimento: Sin dejar de fijar la vista en su monitor intente describir (sin usar su memoria) algún otro objeto de la habitación. Luego separe la vista del Monitor y observe detenidamente este objeto y perciba todos los detalles que se le escaparon en la descripción. Cuando la vista está fija en determinado punto los elementos que están más lejanos en el ángulo de visión suelen lucir desenfocados. La Figura 9-16 se acerca también a esta sensación al dejar fuera de foco todos los elementos excepto el Border y su contenido que representan al mensaje que se pretende que sea el centro de atención cuando sucede el error. El efecto se hace más evidente si el desenfoque se aplica de manera progresiva desde el centro hacia fuera de modo que lo más lejano al centro de atención sea lo que tenga mayor desenfoque. La Figura 9-18 muestra el resultado de aplicar este desenfoque progresivo pero le recomendamos que sólo lo emplee en ventanas grandes o a pantalla completa para que luzca aún más real.

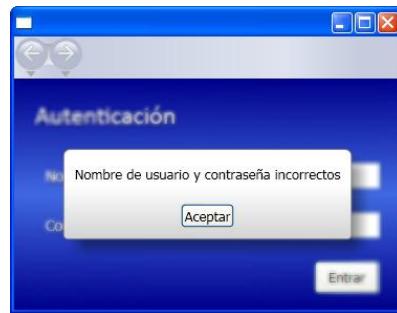


Figura 9-18 Efecto de enfoque/desenfoque con BlurBitmapEffectEffect

Materiales cristalinos: Hay materiales cristalinos que también provocan una imagen de desenfoque cuando vemos a través de ellos. Tome algún objeto de cristal grueso y compare la calidad de la imagen que ve a través de él con la imagen que tendría si mirara al mismo lugar esta vez sin el cristal de por medio. Notará que con el objeto cristalino ocurre un ligero desenfoque en la imagen de fondo.

Cristales empañados: Este es un efecto parecido al anterior pero se basa en que el efecto de desenfoque no se aplica a toda el área que cubre el elemento, de modo que la imagen resultante hace parecer que se ha removido la humedad en una parte de un cristal empañado (Figura 9-19).

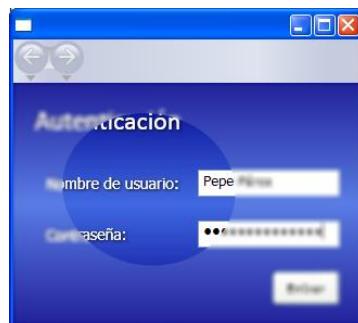


Figura 9-19 Ventana con efecto de Cristal empañado

9.2.4 Efecto Relieve

Centrémonos nuevamente en el título de la ventana de Autenticación cuando no tenía efecto de Bitmap alguno (Figura 9-20).



Figura 9-20 Título de ventana de autenticación sin efectos de Bitmap

Observe de nuevo el aspecto plano del texto sobre la ventana. El relieve es uno de los efectos que nos da un carácter tridimensional de los elementos. Si le aplicáramos a este texto un efecto de EmbossBitmapEffect (Efecto de Relieve) como en el Listado 9-15 obtendríamos un texto con más cuerpo que cuando usamos sombra (Figura 9-21).



Figura 9-21 Texto con efecto de relieve

Listado 9-15 TextBlock con efecto de EmbossBitmapEffect

```
<Page x:Class="VisualEffects.Emboss"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Emboss">
    <Grid>
        <Grid.Background>
            <LinearGradientBrush EndPoint="1,1">
                <GradientStop Color="Navy" Offset="0"/>
                <GradientStop Color="RoyalBlue" Offset="1"/>
            </LinearGradientBrush>
        </Grid.Background>
        <TextBlock VerticalAlignment="Top" HorizontalAlignment="Left"
            Margin="20" Foreground="White"
            FontFamily="Calibry" FontSize="26">
            Autenticación
            <TextBlock.BitmapEffect>
                <EmbossBitmapEffect Relief="1" LightAngle="135"/>
            </TextBlock.BitmapEffect>
        </TextBlock>
    </Grid>
</Page>
```

Compare la Figura 9-20 y la Figura 9-21 y observe que en esta última el texto luce como levantado con respecto del fondo. Esto se debe al ángulo que hemos dado a la luz con LightAngle.

El efecto de relieve se aplica sobre todo a los contornos de las figuras dentro de la imagen en correspondencia con la dirección en que se aplica la luz, de modo que mientras más lejos estén

estos bordes del centro de luz estos luzcan más oscuros. La suavidad de la degradación depende de la propiedad Relief.

9.2.5 Efecto Bisel

El Bisel o BevelBitmapEffect es un efecto parecido al EmbossBitmapEffect. Su función es también la de aportar relieve (levantar o hundir) a los elementos pero este se concentra en la figura que determina al elemento al que se está aplicando el efecto. Tomemos como ejemplo la estrella que se obtuvo en la Lección **Brochas** cuando rellenamos con un gradiente lineal al polígono que describe una estrella (Figura 9-22).



Figura 9-22 Estrella obtenida en la Lección Figuras y en la Lección Brochas

Aunque con la brocha de gradiente lineal se puede reducir el carácter plano de las brochas sólidas. Probablemente usted aún no esté conforme. Esta estrella todavía puede tomar más cuerpo. Si aplicáramos un efecto de bisel podríamos levantar a la estrella desde sus bordes (Listado 9-16 y Figura 9-23).

Listado 9-16 Polígono con efecto de BevelBitmapEffect

```
<Page x:Class="VisualEffects.Bevel"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Bevel">
    <Grid>
      <Polygon Points="10,40 32,40 40,10 48,40 70,40 53,53 60,80 40,65 20,80
27,53"
        Stroke="DarkGoldenrod">
        <Polygon.Fill>
          <LinearGradientBrush EndPoint="0,1">
            <GradientStop Color="LightYellow" Offset="0"/>
            <GradientStop Color="Yellow" Offset="0.4"/>
            <GradientStop Color="Gold" Offset="0.6"/>
            <GradientStop Color="Goldenrod" Offset="1"/>
          </LinearGradientBrush>
        </Polygon.Fill>
        <Polygon.BitmapEffect>
          <BevelBitmapEffect BevelWidth="2" Relief="0.2"
```

```

    LightAngle="135" Smoothness="0.8"/>
</Polygon.BitmapEffect>
</Polygon>
</Grid>
</Page>

```



Figura 9-23 Estrella levantada con efecto de Bisel

Este efecto puede cubrir más el interior de la figura si aumentamos el ancho del bisel: BevelWidth = "12" (Figura 9-24).



Figura 9-24 Efecto de Bisel con ancho aumentado

Si quisieramos ahora que el relieve de la estrella de la Figura 9-24 luzca menos suave, podemos entonces reducir el valor de la propiedad Softness a 0.2 (Figura 9-25).



Figura 9-25 Efecto de Bisel poco suavizado

La propiedad Relief actúa de manera similar a la del efecto Emboss. Esta propiedad regula la suavidad con que ocurre la degradación hacia el obscurecimiento dando impresión de mayor o menor relieve. Haciendo Relief = "0.4" la estrella luciría como en la Figura 9-26.



Figura 9-26 Mayor relieve en la estrella

BevelBitmapEffect cuenta además con una propiedad EdgeProfile que indica el tipo de relieve que se aplica al elemento. La muestra los diferentes valores de esta propiedad.

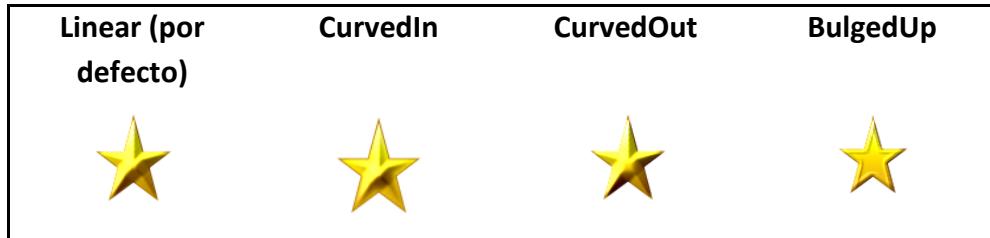


Figura 9-27 Diferentes valores de la propiedad EdgeProfile

En el valor BulgedUp de la Figura 9-27 hemos reducido el ancho del Bisel para hacer notar mejor el efecto.

Recuerde usted que estos efectos todos estos efectos son aplicables a cualquier elemento de WPF. A modo de ejemplo hemos puesto un ComboBox en el Listado 9-17 con un efecto de Bisel similar al que hemos usado en la estrella con EdgeProfile = "CurvedOut" que se muestra en la Figura 9-28.

Listado 9-17 ComboBox con BevelBitmapEffect

```
<Page x:Class="VisualEffects.Bevel"

      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Bevel">
    <Grid>
      <Polygon ...>
        ...
        <Polygon.BitmapEffect>
          <BevelBitmapEffect      BevelWidth="12"      Relief="0.4"
            LightAngle="135"
            Smoothness="0.2" EdgeProfile="CurvedIn"/>
        </Polygon.BitmapEffect>
      </Polygon>
      <ComboBox Margin="10" Width="130" Padding="2,1"
                VerticalAlignment="Bottom" HorizontalAlignment="Left">
        Curvo hacia fuera
        <ComboBox.BitmapEffect>
          <BevelBitmapEffect      BevelWidth="12"      Relief="0.4"
            LightAngle="135"
            Smoothness="0.2" EdgeProfile="CurvedOut"/>
        </ComboBox.BitmapEffect>
      </ComboBox>
    </Grid>
```

```
</Page>
```



Figura 9-28 ComboBox con efecto de Bisel

9.2.6 Grupos de efectos

Como usted habrá notado cada uno de los efectos descritos anteriormente puede aportar mucho realismo a los elementos que usted emplee en su aplicación. Sin embargo, con WPF usted puede aspirar a más... Observe la Figura 9-28 e imagínese ahora la estrella levantada del fondo. Como vimos anteriormente esto se puede lograr fácilmente con un efecto de sombra, pero ya la estrella ya tiene aplicado un efecto de Bitmap (BevelBitmapEffect). Empleando los grupos de efectos (BitmapEffectGroup) se puede aplicar más de un efecto al mismo elemento. En el Listado 9-18 hemos usado un grupo de efectos para añadir también una sombra a la estrella de la Figura 9-28. Observe el resultado en la Figura 9-29.

Listado 9-18 Polígono con un grupo de efectos de Bitmap

```
<Page ...>

<Grid>
    <Polygon ...>
        ...
        <Polygon.BitmapEffect>
            <BitmapEffectGroup>
                <BevelBitmapEffect      BevelWidth="12"      Relief="0.4"
LightAngle="135"
                    Smoothness="0.2" EdgeProfile="CurvedIn"/>
                <DropShadowBitmapEffect Softness="0.4"      Color="Black"
Opacity="0.7"
                    Direction="315" ShadowDepth="7"/>
            </BitmapEffectGroup>
        </Polygon.BitmapEffect>
    </Polygon>
</Grid>
</Page>
```



Figura 9-29 Estrella con efectos de bisel y sombra

Tenga cuidado con el orden de empleo de estos los grupos de efectos porque cada efecto interno se aplica siempre sobre el Bitmap resultante de aplicar el anterior. En el Listado 9-19 hemos añadido un efecto de mucho desenfoque a la estrella. Observe cómo la sombra también se afecta con este efecto en la Figura 9-30.

Listado 9-19 Polígono con tres efectos de bitmap

```
<Page ...>
  <Grid>
    <Polygon ...>
      ...
      <Polygon.BitmapEffect>
        <BitmapEffectGroup>
          <BevelBitmapEffect BevelWidth="12" Relief="0.4" LightAngle="135"
                            Smoothness="0.2" EdgeProfile="CurvedIn"/>
          <DropShadowBitmapEffect Softness="0.4" Color="Black"
                                 Opacity="0.7"
                                 Direction="315" ShadowDepth="7"/>
          <BlurBitmapEffect Radius="5"/>
        </BitmapEffectGroup>
      </Polygon.BitmapEffect>
    </Polygon>
  </Grid>
</Page>
```



Figura 9-30 Estrella con efectos de bisel, sombra y desenfoque aplicados en ese orden

Si ahora invirtiéramos el orden de modo que el Desenfoque (BlurBitmapEffect) apareciera antes que los demás no obtendríamos el efecto deseado (Figura 9-31).



Figura 9-31 Estrella con efectos de desenfoque, bisel y sombra aplicados en ese orden

9.2.7 Rendimiento

Aunque cada uno de estos efectos tiene un trasfondo implementado nativamente que los hace aplicarse con bastante rapidez, a continuación se exponen algunos aspectos a tener en cuenta ya que la aplicación de estos efectos pudiera afectar el rendimiento de su aplicación:

Empleo de los grupos de efectos

Como indicamos antes los grupos de efectos hacen que cada efecto se aplique sobre el Bitmap resultante de aplicar el efecto anterior. Si el efecto que usted está aplicando está siendo aplicado a un elemento de grandes dimensiones el tratamiento de los Bitmaps que se generan uno tras otro por cada efecto puede disminuir el rendimiento de su aplicación.

Empleo inapropiado de sombra y resplandor en elementos opacos y dinámicos

Por regla general los efectos del Resplandor y Sombra suelen notarse en la parte externa del elemento que se le aplica. Solamente cuando el elemento es transparente o semitransparente usted notará la aplicación de estos efectos en el interior del elemento. Observe la Figura 9-32 donde mostramos una ventana azul que contiene un Grid transparente al que se le ha aplicado un efecto de sombra.

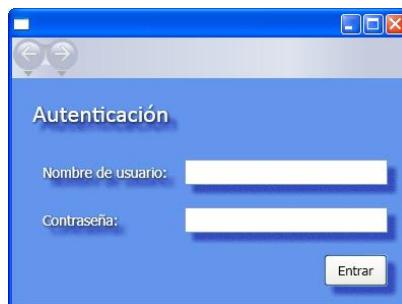


Figura 9-32 Efecto de sombra aplicado a un Grid transparente

En la Figura 9-32 se observa cómo el efecto de sombra actúa cuando el elemento es transparente. En casos como este la sombra aparece aplicada a todos los elementos opacos internos. No hay inconvenientes si es esto lo que usted espera, aunque sugeriríamos que aplicara el efecto a cada uno de los elementos en vez de al Grid completo. Todos los efectos de Bitmap se aplican siempre a una imagen que se genera a partir de la definición del elemento y sus elementos internos. Si por ejemplo, usted aplica un efecto de sombra sobre un Border que

a su vez contenga un TextBox, el efecto de sombra será vuelto a calcular y pintar cada vez que usted escriba una letra en dicho TextBox; porque modificar el texto siempre origina un cambio en la imagen a la que se aplicó la sombra. El chiste está en que si el Border y el TextBox son opacos, el recálculo y el repintado no son notables porque la sombra se ve fuera del Border y la nueva sombra entonces luce idéntica a la que se tenía antes del cambio de texto en el TextBox.

Si el elemento contenedor y el propio TextBox fueran opacos usted tiene una forma de evitar que estas operaciones gráficas se repitan innecesariamente: En lugar de aplicar la sombra al Border que contiene al TextBox usted puede poner donde mismo está el Border un Grid que contenga otro Border idéntico al primero, **sin contenido** y **con el efecto de sombra**, para luego poner encima el Border que contiene al TextBox **sin sombra**. Hecho de esta manera cuando se escriba sobre el TextBox, WPF no recalcula ni repinta la sombra porque esta no se corresponde con el elemento que contiene al TextBox. Observe que este truco es útil sólo cuando los elementos son opacos porque en un TextBox semitransparente con sombra, el texto también tiene sombra y esta sí debe ser recalculada y repintada al cambiar el texto.

Aclaremos que estas consideraciones son aplicables para todos los efectos que se utilicen en elementos que convenga que cambien con frecuencia de aspecto (como por ejemplo un TextBox). Si hemos insistido en los efectos de sombra y resplandor es porque estos son los efectos de Bitmap que más se utilizan en elementos como estos. ¿No le resulta poco realista escribir sobre un TextBox desenfocado?

Lección 10 Transformaciones

Basados en el sistema de layout de WPF hay dos funcionalidades que permiten realizar transformaciones a los elementos que conforman una interfaz, las transformaciones se basan en el cambio de posición de todos los puntos de coordenadas de un elemento gráfico. Una transformación se logra dándole valor a las propiedades LayoutTransform o RenderTransform de los elementos visuales. Las transformaciones están disponibles en el espacio de nombres System.Windows.Media y tienen como base la clase Transform.

Cualquiera de las transformaciones definidas en WPF puede aplicarse sobre cualquier elemento de interfaz de usuario (UIElement). A un elemento se le puede aplicar cualquier cantidad de transformaciones. La clase TransformGroup (derivada de Transform) representa una composición de transformaciones. Contiene una colección de Transform.

Si la transformación es aplicada como RenderTransform (asignándole la transformación a la propiedad RenderTransform de un UIElement), esta afectaría la posición de renderizado del

elemento. Si son aplicadas a la propiedad LayoutTransform (de FrameworkElement), estaría afectando el resultado del layout. Uno de los objetivos de esta lección es que Ud. comprenda la diferencia entre ambas y cuando es propicio usar una o la otra.

Si Ud. tiene alguna experiencia previa con transformaciones gráficas (en Win32 o Windows Forms por ejemplo), notará algunas diferencias en la forma en la que las transformaciones son implementadas en WPF. En otras tecnologías para desarrollo gráfico precedentes las transformaciones son propiedades de la superficie donde se “pinta” la interfaz, y todo lo que se “pinte” sobre esta superficie estará sujeto a las transformaciones. En WPF, las transformaciones son propiedades de los elemento visuales, y se realizan de manera efectiva relativas al elemento en si mismo.

Por ejemplo, para rotar un elemento visual en WPF todo lo que debemos especificar es un ángulo de rotación, en otros ambientes gráficos convencionales esta rotación se centraría alrededor la superficie donde se pinta (por ejemplo el punto (0,0) de un Canvas donde está contenido el elemento). Sin embargo en WPF la rotación puede especificarse como relativa a un punto en el propio elemento.

A partir del ejemplo del Listado 10-1 comenzaremos a realizar transformaciones mostrando el resultado de la ejecución en las correspondientes figuras. Ilustramos primero las transformaciones que son de tipo RenderTransform y posteriormente las que son de tipo LayoutTransform para que Ud. note la diferencia entre una y otra. Aunque ambas son similares en el hecho de que son de tipo Transform, podemos adelantarle que la primera no respeta la distribución de layout que se le da a la interfaz visual mientras la segunda sí tiene en cuenta este aspecto. En esencia con LayoutTransform la transformación aplica también al posicionamiento de los elementos de layout, mientras que con RenderTransform sólo afecta el renderizado de los elementos visuales. Cuando se realiza un LayoutTransform sobre un elemento este se transforma en el área que le corresponde según el sistema de layout aplicado.

En este ejemplo hemos realizado las transformaciones sobre un control Image, pero de igual manera son válidas para cualquier elemento visual de WPF.

Listado 10- 1 Código XAML ventana sin transformaciones

```
<Window x:Class="TransformationSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Transformaciones"           Height="300"           Width="300"
Background="DarkBlue"
>
<StackPanel>
```

```

<Image Source="Windito.gif" Height="200"/>
<TextBlock HorizontalAlignment="Center" FontFamily="Arial"
FontSize="30" FontWeight="Bold" Foreground="White">
"Windito"
</TextBlock>
</StackPanel>
</Window>

```

La Figura 10-1 muestra la ejecución del código del Listado 10-1 sobre el que comenzaremos a aplicar transformaciones.

Para su mejor comprensión, en las figuras que le mostramos en esta lección aparecerá semitransparente la imagen en su posición original (antes de aplicar cada transformación), de esta forma Ud. puede constatar mejor el resultado de una determinada transformación. Aunque este efecto fue programado a su vez en el código XAML de los ejemplos, no se ha incluido en los listados para no distraer su atención más allá del objetivo de esta lección.



Figura 10- 1 Ventana sin transformaciones.

WPF tiene cinco clases para hacer transformaciones, estas son `RotateTransform`, `ScaleTransform`, `SkewTransform`, `TranslateTransform` y `MatrixTransform`.

10.1 TranslateTransform

Con esta transformación se traslada un elemento en un espacio bidimensional especificando en las propiedades `X` y `Y` la cantidad en píxeles para trasladar en el eje `x` y en el eje `y`.

Matemáticamente esto puede expresarse como sigue:

(x,y) : posición original del elemento visual.

(tx,ty) : valor de las propiedades `X` y `Y` de `TranslateTransform`.

(x',y') : posición en la que se muestra el elemento luego de aplicada la transformación (posición de renderizado) y puede expresarse:

$x' = x + tx;$

$y' = y + ty$

Note que tanto tx como ty pueden tener valores negativos. El código del Listado 10-2 traslada nuestra imagen 50 píxeles por la horizontal hacia la derecha y 100 píxeles por la vertical hacia abajo

```
Listado 10- 2 Código XAML TranslateTransform como  
RenderTransform
```

```
<StackPanel>  
  <Image Source="Windito.gif" Height="200">  
    <Image.RenderTransform>  
      <TranslateTransform X="50" Y="100"/>  
    </Image.RenderTransform>  
  </Image>  
  <TextBlock HorizontalAlignment="Center" FontFamily="Arial"  
            FontSize="30" FontWeight="Bold" Foreground="White">  
    "Windito"  
  </TextBlock>  
</StackPanel>
```

La Figura 10-2 muestra el resultado de la ejecución de este código. Para que observe la diferencia se ha dejado la imagen semitransparente de fondo en la posición original.



Figura 10- 2 TranslateTransform como RenderTransform.

El eje de coordenadas se ha colocado en la esquina superior izquierda pues ahí comienza la imagen. La imagen es rectangular, lo que sucede es que el fondo de esta se ha hecho transparente para una mejor apariencia visual dando la impresión de que la imagen fuese solo "Windito".

Si Ud. cambia en el código XAML del Listado 10-2 la propiedad `Image.RenderTransform` por `Image.LayoutTransform` y lo ejecuta, notará que al asignar valor a cualquiera de las propiedades `X` y `Y` de `TranslateTransform` para trasladar el elemento, la imagen no se trasladará. A los efectos

visuales de la interfaz esta transformación no será aplicada por respetarse la distribución de layout de los elementos. Es decir, que TranslateTransform aplicado a la propiedad LayoutTransfrom no tienen efecto alguno en la ubicación del elemento.

10.2 RotateTransform

Rota un elemento visual en el plano bidimensional en el ángulo especificado (propiedad Angle) y un punto centro de la rotación expresado en el espacio de coordenadas del elemento que está siendo transformado (propiedades CenterX y CenterY).

Matemáticamente esto puede expresarse como sigue:

(x,y) : posición original del elemento visual.

α : valor de la propiedades Angle de RotateTransform.

(x',y') : posición en la que se muestra el elemento luego de aplicada la trasformación (posición de renderizado) donde:

$$x' = \cos(\alpha) * x - \sin(\alpha) * y$$

$$y' = \sin(\alpha) * x + \cos(\alpha) * y$$

Esta rotación ocurre tomando como centro la esquina superior izquierda del elemento visual al que se aplica la transformación. Es posible indicar el punto centro de la rotación mediante las propiedades CenterX y CenterY, de hecho al dar valor a estas propiedades (suponga que los valores asignados son cx y cy) la fórmula para rotar un elemento quedaría:

$$x' = \cos(\alpha) * (x - cx) - \sin(\alpha) * (y - cy) + cx$$

$$y' = \sin(\alpha) * (x - cx) + \cos(\alpha) * (y - cy) + cy$$

El Listado 10-3 muestra cómo rotar la imagen 45 grados (Figura 10-3)

Listado 10- 3 Código XAML RotateTransform como RenderTransform

```
<StackPanel>
    <Image Source="Windito.gif" Height="200">
        <Image.RenderTransform>
            <RotateTransform Angle="45"/>
        </Image.RenderTransform>
    </Image>
    <TextBlock HorizontalAlignment="Center" FontFamily="Arial"
        FontSize="30" FontWeight="Bold" Foreground="White">
        "Windito"
    </TextBlock>
</StackPanel>
```



Figura 10- 3 RotateTransform como RenderTransform.

Note como no se respeta el layout del elemento al aplicar la transformación (texto ha quedado por encima de la imagen a pesar de que están en un StackPanel). Sin embargo, si esta transformación se hubiese aplicado dentro de un LayoutTransform (Listado 10-4) esto no hubiese ocurrido (Figura 10-4).

Listado 10- 4 Código XAML RotateTransform como LayoutTransform

```
<Image.LayoutTransform>
    <RotateTransform Angle="45"/>
</Image.LayoutTransform>
```

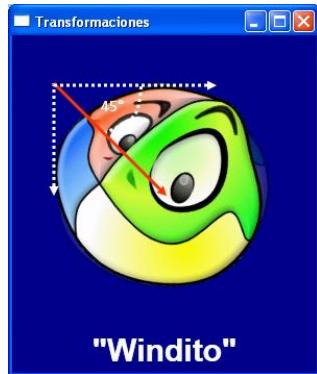


Figura 10- 4 RotateTransform como LayoutTransform.

10.3 ScaleTransform

Esta transformación escala un elemento modificando su tamaño comenzando por un punto centro definido (propiedad `CenterX` y `CenterY`). Valores diferentes para las coordenadas x y y pueden especificarse con las propiedades `ScaleX` y `ScaleY` cuyo valor predeterminado es 1. Por defecto una transformación de escala tiene como centro el punto $(0, 0)$, pero es posible escalar por ejemplo a $(Width/2, Height/2)$.

Matemáticamente esto puede expresarse como sigue:

(x,y) : posición original del elemento visual.

(sx,sy) : valor de las propiedades `ScaleX` y `ScaleY` de `ScaleTransform`.

(x',y') : posición el la que se muestra el elemento luego de aplicada la trasformación (posición de renderizado) y puede expresarse:

$$x' = x * sx$$

$$y' = y * sy$$

¿Qué pasa con las coordenadas del centro del elemento una vez que este ha sido transformado? Pues producto de la transformación el centro habrá cambiado de posición. Es posible especificar que un elemento mantenga el centro en las mismas coordenadas, es decir que la modificación de sus dimensiones ocurra de forma simétrica,

Este es el propósito de las propiedades `CenterX` y `CenterY`, de hecho al dar valor a estas propiedades (suponga que los valores asignados son cx y cy) la fórmula para escalar un elemento quedaría:

$$x' = sx * (x - cx) + cx = sx * x + (cx - sx * cx)$$

$$y' = sy * (y - cy) + cy = sx * y + (cy - sy * cy)$$

Para escalar un elemento al 200 % por las coordenadas en el eje de las **y**, pero no por las **x** (como ilustra la Figura 10-5) use el objeto ScaleTransform como se muestra en el Listado 10-5.

Listado 10- 5 Código XAML ScaleTransform como RenderTransform

```
<StackPanel>
    <Image Source="Windito.gif" Height="200">
        <Image.RenderTransform>
            <ScaleTransform ScaleY="2"/>
        </Image.RenderTransform>
    </Image>
    <TextBlock HorizontalAlignment="Center" FontFamily="Arial"
        FontSize="30" FontWeight="Bold" Foreground="White">
        "Windito"
    </TextBlock>
</StackPanel>
```



Figura 10- 5 ScaleTransform como RenderTransform.

Veamos en el Listado 10-6 la misma transformación pero asignada a la propiedad LayoutTransform. La Figura 10-6 muestra el resultado de la ejecución.

Listado 10- 6 Código XAML ScaleTransform como LayoutTransform

```
<Image.LayoutTransform>
    <ScaleTransform ScaleY="2"/>
</Image.LayoutTransform>
```

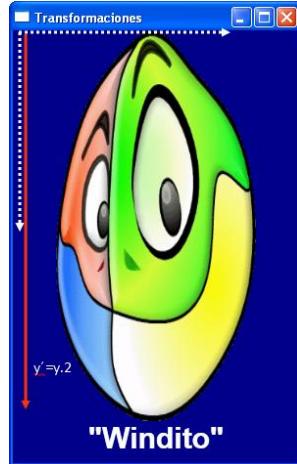


Figura 10-6 ScaleTransform como LayoutTransform.

Uno de los usos más comunes de ScaleTransform es lograr un efecto de ampliación (zoom) a un elemento. Veamos por ejemplo, como agrandar una imagen al doble de su tamaño (Listado 8-7).

Listado 10- 7 Código XAML Zoom con ScaleTransform

```
<StackPanel>
    <Image Source="Windito.gif" Height="200">
        <Image.RenderTransform>
            <ScaleTransform ScaleX="2" ScaleY="2"/>
        </Image.RenderTransform>
    </Image>
    <TextBlock HorizontalAlignment="Center" FontFamily="Arial"
        FontSize="30"          FontWeight="Bold"
        Foreground="White">
        "Windito"
    </TextBlock>
</StackPanel>
```

La Figura 10-7 a continuación muestra el resultado del efecto de ampliación sobre la imagen.



Figura 10- 7 Zoom con ScaleTransform.

Los valores de las propiedades ScaleX y ScaleY pueden ser negativos, por ejemplo para el caso ScaleX=-1, ocurrirá que el elemento visual se “volteará” sobre su lado izquierdo provocando la sensación de una imagen reflejo. Vea Listado 10-8 y Figura 10-8.

Listado 10- 8 Código XAML Reflejo horizontal con ScaleTransform

```
<StackPanel VerticalAlignment="Center">
    <Image Source="Windito.gif" Height="200">
        <Image.RenderTransform>
            <ScaleTransform ScaleY="-1"/>
        </Image.RenderTransform>
    </Image>
    <TextBlock HorizontalAlignment="Center" FontFamily="Arial"
        FontSize="30" FontWeight="Bold" Foreground="White">
        "Windito"
    </TextBlock>
</StackPanel>
```



Figura 10- 8 Reflejo horizontal con ScaleTransform.

También podemos hacer ScaleY=-1 y el elemento se “voltea” sobre su arista superior. Haciendo uso de las escalas negativas de un elemento podemos lograr un efecto reflejo. Vea Listado 10-9 y Figura 10-9.

Listado 10- 9 Código XAML Reflejo vertical con ScaleTransform

```
<StackPanel VerticalAlignment="Center">
    <Image Source="Windito.gif" Height="200">
        <Image.RenderTransform>
            <ScaleTransform ScaleX="-1"/>
        </Image.RenderTransform>
    </Image>
    <TextBlock HorizontalAlignment="Center" FontFamily="Arial"
        FontSize="30" FontWeight="Bold" Foreground="White">
        "Windito"
    </TextBlock>
</StackPanel>
```



Figura 10- 9 Reflejo vertical con ScaleTransform.

10.4 SkewTransform

Esta transformación define una inclinación en dos dimensiones que estira el sistema de coordenadas en una manera no uniforme. Las propiedades CenterX y CenterY especifican el punto centro para la transformación y tienen valor (0,0) por defecto. AngleX y AngleY el ángulo de inclinación por la coordenada *x* y la *y*, tienen valor (0,0) por defecto. La transformación realiza la inclinación a partir de los valores de *x* y *y* relativos al sistema de coordenadas original.

Matemáticamente esto puede expresarse como sigue:

(*x,y*): posición original del elemento visual.

(α_x, α_y): valor de las propiedades AngleX y AngelY de SkewTransform.

(x',y') : posición en la que se muestra el elemento luego de aplicada la transformación (posición de renderizado) y puede expresarse:

$$x' = x + \tan(\alpha x * y)$$

$$y' = y + \tan(\alpha y * x)$$

También es posible especificar el centro de la transformación mediante las propiedades CentreX y CenterY (suponga que los valores asignados son cx y cy) de SkewTransform, luego la fórmula quedaría:

$$x' = x + \tan(\alpha x * y) * (y - cx)$$

$$y' = y + \tan(\alpha y * x) * (x - cy)$$

Para inclinar un elemento 45 grados por las **y**, se debe usar SkewTransform como se muestra en el Listado 10-10

Listado 10-10 Código XAML SkewTransform como RenderTransform

```
<StackPanel>
    <Image Source="Windito.gif" Height="200">
        <Image.RenderTransform>
            <SkewTransform AngleY="45"/>
        </Image.RenderTransform>
    </Image>
    <TextBlock HorizontalAlignment="Center"
        FontFamily="Arial"
        FontSize="30"
        FontWeight="Bold"
        Foreground="White">
        "Windito"
    </TextBlock>
</StackPanel>
```

La Figura 10-10 muestra el resultado de la ejecución del código XAML del Listado 10-10.

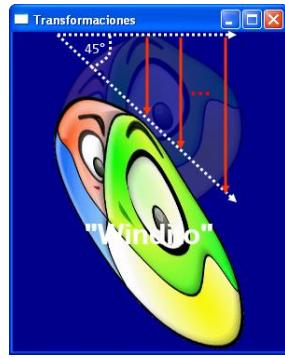


Figura 10- 10 SkewTransform como RenderTransform.

Veamos ahora la misma transformación aplicada a [LayoutTransform](#) (Listado 10-11).

Listado 10- 11 Código XAML SkewTransform como LayoutTransform

```
<Image.LayoutTransform>  
  <SkewTransform AngleY="45"/>  
</Image.LayoutTransform>
```

Veamos el resultado de la ejecución en la Figura 10-11.

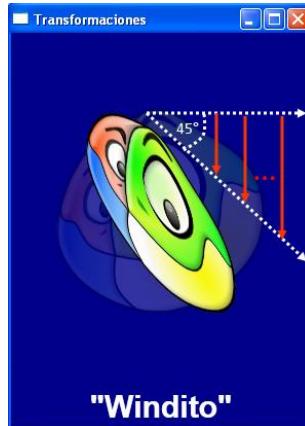


Figura 10- 11 SkewTransform como LayoutTransform.

10.5 MatrixTransform

Con esta forma de transformación se puede manipular un elemento en el espacio bidimensional permitiendo realizar transformaciones que no han sido incluidas predeterminadas como las que hemos visto anteriormente.

Existen múltiples transformaciones posibles a realizar haciendo uso de MatrixTransform, no es nuestro objetivo entrar en detalles de todo lo que Ud. puede hacer, más bien lograr atraer su interés mostrando los detalles básicos a conocer e intentar motivarlo a probar.

La transformación que podemos lograr con Matrixtransform es una transformación lineal. Esta transformación es descrita usando como base la representación matemática de una matriz. Ilustraremos esto usando MatrixTransform para hacer alguna de las transformaciones ya vistas en esta lección pero esta vez usando MatrixTransform.

Un punto, x, y es trasladado a un nuevo punto, x', y' usando la siguiente formula:

$$x' = x * M11 + y * M21 + \text{OffsetX}$$

$$y' = x * M12 + y * M22 + \text{OffsetY}$$

Para usar una tal matriz hay que escribir en XAML:

```
<MatrixTransform Matrix="M11, M12, M21, M22, OffsetX, OffsetY"/>
```

o lo que es lo mismo (poniendo entre las comillas los valores que se quieran asignar):

```
<MatrixTransform>
  <MatrixTransform.Matrix>
    <Matrix M11="" M12="" M21="" M22="1" OffsetX="" OffsetY="" />
  </MatrixTransform.Matrix>
</MatrixTransform>
```

Con esta información es relativamente sencilla la generación de transformaciones simples. Por ejemplo escalar una imagen al doble de su tamaño en la dirección de las x quedaría:

```
<Image.RenderTransform>
  <MatrixTransform>
    <MatrixTransform.Matrix>
      <Matrix M11="2" M12="0" M21="0" M22="1" OffsetX="0" OffsetY="0" />
    </MatrixTransform.Matrix>
  </MatrixTransform>
</Image.RenderTransform>
```

O trasladar una imagen 100 píxeles en la dirección de las x y y :

```
<Image.RenderTransform>
  <MatrixTransform>
    <MatrixTransform.Matrix>
      <Matrix M11="1" M12="0" M21="0" M22="1" OffsetX="100" OffsetY="100" />
    </MatrixTransform.Matrix>
```

```
</MatrixTransform>  
</Image.RenderTransform>
```

O en otro caso lograr un efecto de reflejo (rotar alrededor de las x):

```
<Image.RenderTransform>  
<MatrixTransform>  
  <MatrixTransform.Matrix>  
    <Matrix M11="1" M12="0" M21="0" M22="-1" OffsetX="0" OffsetY="0"/>  
  </MatrixTransform.Matrix>  
</MatrixTransform>  
</Image.RenderTransform>
```

10.6 TransformGroup

Las transformaciones pueden agruparse para aplicarse en conjunto usando TransformGroup. El código del Listado 10-12 es un ejemplo de dos transformaciones: una que primero rota y otra inclina la imagen (Figura 10-12) agrupadas en un TransformGroup.

Listado 10- 12 Código XAML TransformGroup

```
<StackPanel>  
  <Image Source="Windito.gif" Height="200">  
    <Image.RenderTransform>  
      <TransformGroup>  
        <RotateTransform Angle="45"/>  
        <SkewTransform AngleY="45"/>  
      </TransformGroup>  
    </Image.RenderTransform>  
  </Image>  
  <TextBlock HorizontalAlignment="Center" FontFamily="Arial"  
            FontSize="30" FontWeight="Bold" Foreground="White">  
    "Windito"  
  </TextBlock>  
</StackPanel>
```

En la Figura 10-12 se muestran los distintos estados (imágenes semitransparentes al fondo) por los que transita el grupo de transformaciones para obtener el resultado final (claro que en la ejecución del Listado 10-12 esto no se apreciará).



Figura 10- 12 TransformGroup.

Transformando el contenido de una brocha

Por último, pero no por ello menos útil, veremos una forma de aplicar transformaciones que es distinta a las que habíamos visto antes en esta misma lección (LayoutTransform y RenderTransform). Esta forma de aplicar transformaciones es específica para el tipo Brush donde las transformaciones se aplican a través de las propiedades Transform y RelativeTransform de tipo Transform.

Estas propiedades permiten *trasladar, rotar, escalar e inclinar* el contenido de una brocha. Cuando se aplica una transformación a través de la propiedad Transform las coordenadas para la transformación son relativas al área que se va a "pintar" por lo que es necesario conocer sus dimensiones. Con la propiedad RelativeTransform las transformaciones a aplicar ocurren relativas al elemento brocha, y se aplican antes de que ésta sea usada por WPF para "pintar" una determinada superficie.

El elemento que define el contenido de una brocha se encuentra enmarcado en un rectángulo imaginario de 1x1 como el que muestra la Figura 10-13. Relativo a sus dimensiones es que se aplican las transformaciones sobre la brocha.

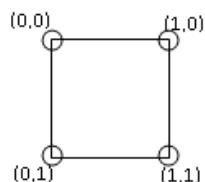


Figura 10- 13 Rectángulo imaginario usado por una brocha para calcular el área a colorear

Veamos como ejemplo el Listado 10-13 que muestra cómo lograr un efecto de reflejo de una imagen aplicando transformaciones relativas a una brocha. Note en el Listado 10-13 la aplicación de una transformación de escala (ScaleTransform) relativa a la brocha con que luego se rellena un rectángulo. En este caso al especificar CenterY="0.5", estamos situando el valor de la propiedad CenterY en el punto medio del rectángulo de transformaciones que contiene al elemento que conforma la brocha. De esta forma luego de aplicada la transformación, el

elemento brocha mantiene los valores de su punto medio (solo es necesario especificar CenterY pues la transformación es aplicada en el sentido del eje "y").

Listado 10- 13 Código XAML RelativeTransform para el efecto reflejo

```
<StackPanel>
    <Image Source="Windito.gif" Name="Windito" Height="100"/>
        <Rectangle Width="{Binding ElementName=Windito, Path=ActualWidth}"
                   Height="{Binding ElementName=Windito, Path=ActualHeight}">
            <Rectangle.OpacityMask>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0" Color="#FF000000"/>
                        <GradientStop Offset="0.8" Color="#00000000"/>
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Rectangle.OpacityMask>
            <Rectangle.Fill>
                <VisualBrush Visual="{Binding ElementName=Windito}">
                    <VisualBrush.RelativeTransform>
                        <ScaleTransform ScaleY="-1" CenterY="0.5"/>
                    </VisualBrush.RelativeTransform>
                </VisualBrush>
            </Rectangle.Fill>
        </Rectangle>
    </StackPanel>
```

La Figura 10-13 a continuación muestra el resultado de la ejecución de éste código.



Figura 10- 13 RelativeTransform de Brush para lograr un efecto de reflejo.

Capítulo IV ESTILOS Y PLANTILLAS

Este capítulo está dedicado a estilos y plantillas. Los estilos nos permiten personalizar la apariencia de la interfaz de usuario de modo similar y mas amplio que lo que por ejemplo se puede hacer con los estilos en Microsoft Word. Las plantillas (*templates*) son un recurso que nos facilitará expresar patrones de código a aplicar a datos y a controles propiciando la reutilización y aumentando la productividad y flexibilidad. Se incluye también una lección sobre triggers (desencadenadores) lo que nos permitirá expresar de forma declarativa lo que queremos que ocurra cuando se cumplan determinadas condiciones en la interfaz de nuestra aplicación.

Lección 11 Estilos

Los controles, figuras, imágenes, y el texto que se muestran en una ventana deben presentarse con un estilo uniforme, agradable e intuitivo, que facilite y estimule la interacción de los usuarios con la aplicación. Por ejemplo, es una buena práctica que las etiquetas que se pongan en una ventana tengan todo un mismo tipo de letra (fuente), color y tamaño de letra. Los estilos sirven para establecer configuraciones uniformes de los elementos que se usan en una interfaz de usuario.

11.1 Personalizando sin estilos

En la Figura 11-1 se muestra una ventana que contiene etiquetas, cajas de contraseña y botones con el estilo predeterminado de WPF.



Figura 11-1 Ventana de cambio de contraseña con estilos predeterminados de WPF

El Listado 11-1 es el XAML correspondiente:

Listado 11-1 Cambio de contraseña
<Window x:Class="StylesIntro.Window1" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

```

Title="Ejemplo de estilos" Height="399" Width="445">
<Window.Background>
  <LinearGradientBrush EndPoint="0,0,5" SpreadMethod="Reflect">
    <GradientStop Color="Navy" Offset="0"/>
    <GradientStop Color="RoyalBlue" Offset="1"/>
  </LinearGradientBrush>
</Window.Background>
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Label Grid.ColumnSpan="2" Margin="20,5" FontSize="20"
        HorizontalAlignment="Left" VerticalAlignment="Top">
    Cambio de contraseña
  </Label>
  <Label Margin="20,5" Grid.Row="1"
        HorizontalAlignment="Left" VerticalAlignment="Top" >
    Contraseña anterior:
  </Label>
  <Label HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="20,5" Grid.Row="2">
    Nueva contraseña:
  </Label>
  <Label HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="20,5" Grid.Row="3">
    Confirmar contraseña:
  </Label>
  <PasswordBox Grid.Column="1" Grid.Row="1" Margin="10,5"/>
  <PasswordBox Grid.Column="1" Grid.Row="2" Margin="10,5"/>
  <PasswordBox Grid.Column="1" Grid.Row="3" Margin="10,5"/>

  <StackPanel Grid.ColumnSpan="2" Grid.Row="4" Margin="10"
             HorizontalAlignment="Center" VerticalAlignment="Top"
             Orientation="Horizontal">
    <Button Margin="10,5">Aceptar</Button>
    <Button Margin="10,5">Cancelar</Button>
  </StackPanel>
</Grid>
</Window>

```

Si ahora quisieramos cambiar las etiquetas para mostrarlos con fuente Candara, 17pt, se podría hacer aplicando el cambio manualmente en el código de cada uno de los elementos `Label` como se muestra a continuación:

Listado 11-2 Cambio de letra a Candara, 17pt.

```
<Window x:Class="StylesIntro.Window1" ...
    Title="Cambio de Contraseña" Height="399" Width="445">
    <Grid>
        ...
        <Label Grid.ColumnSpan="2" FontFamily="Candara" FontSize="20"
            HorizontalAlignment="Left" VerticalAlignment="Top">
            Cambio de contraseña
        </Label>
        <Label Margin="20,5" Grid.Row="1" FontFamily="Candara"
FontSize="17"
            HorizontalAlignment="Left" VerticalAlignment="Top" >
            Contraseña anterior:
        </Label>
        <Label Margin="20,5" Grid.Row="2" FontFamily="Candara"
FontSize="17"
            HorizontalAlignment="Left" VerticalAlignment="Top" >
            Nueva contraseña:
        </Label>
        <Label Margin="20,5" Grid.Row="3" FontFamily="Candara"
FontSize="17"
            HorizontalAlignment="Left" VerticalAlignment="Top">
            Confirmar contraseña:
        </Label>
        ...
    </Grid>
</Window>
```

Note que a diferencia de la Figura 11-1 en la Figura 11-2 la letra de las etiquetas es distinta y su tamaño se ha reducido.



Figura 11-2 Ventana de la Figura 11-1 con las letras cambiadas.

Sin embargo, tras este cambio de fuente ahora las etiquetas apenas se pueden leer. Si en consecuencia también queremos ahora cambiar el tamaño de las letras tendría que aplicarse y

repetirse esto en cada una de los Labels. Esto es una rutina que suele ser bastante tediosa y propensa a equivocaciones, lo que induce a veces a que no se mejoren las apariencias para no pasar este trabajo.

Es común que durante la personalización de una apariencia se apliquen varios cambios continuos sobre un mismo elemento visual hasta dar con una apariencia que nos satisfaga. Imagine lo que significa y hacer directamente los cambios en cada lugar en que se use un elemento visual similar. Los estilos nos van a ayudar en esta labor, al permitirnos configurar una apariencia que luego se pueda aplicar a varios elementos visuales.

11.2 Personalizando con estilos

Los estilos permiten crear configuraciones que indiquen características visuales de controles, figuras, texto, etc. para que luego puedan ser aplicadas a diferentes elementos. Los estilos de WPF son objetos de tipo Style que se recomiendan poner en los recursos de la aplicación.

Si Ud. tiene alguna experiencia de trabajo con Microsoft Word seguro ha disfrutado ya de las bondades de trabajar con estilos. Los estilos en XAML nos dan la posibilidad de generalizar una idea similar a prácticamente cualquier elemento XAML

En el ejemplo del Listado 11-2 se cambió el tipo y tamaño de letra en cada una de las etiquetas. Estos cambios pueden aplicarse definiendo un estilo que se ponga como recurso de la ventana (el elemento Windows que contiene al Grid donde están las etiquetas a las que se va a aplicar el estilo) como se ilustra en el Listado 11-3.

Un estilo puede ponerse como recurso de cualquier elemento del árbol XAML y en tal caso se aplicará a los elementos contenidos. En el ejemplo del Listado 11-3 podía haberse puesto como recurso del Grid y haberse obtenido el mismo efecto.

Listado 11-3 Estilo aplicado a los Labels

```
<Window x:Class="StylesIntro.Window1" ...
       Title="Cambio de Contraseña" Height="399" Width="445">
  <Window.Resources>

    <Style TargetType="{x:Type Label}">
      <Setter Property="FontFamily" Value="Candara"/>
      <Setter Property="FontSize" Value="10"/>
      <Setter Property="HorizontalAlignment" Value="Left"/>
      <Setter Property="VerticalAlignment" Value="Top"/>
      <Setter Property="Margin" Value="20,5"/>
    </Style>
  </Window.Resources>
  <Grid>
    ...
    <Label Grid.ColumnSpan="2" FontSize="20">
      Cambio de contraseña
    </Label>
  </Grid>
</Window>
```

```

</Label>
<Label Grid.Row="1">
    Contraseña anterior:
</Label>
<Label Grid.Row="2">
    Nueva contraseña:
</Label>
<Label Grid.Row="3">
    Confirmar contraseña:
</Label>
...
</Grid>
</Window>

```

Como resultado al ejecutar este XAML se obtendrá una ventana con la misma apariencia de la de Figura 11-2. Note que ahora el XAML de las etiquetas sólo contiene la información propia de cada una, lo demás ha quedado factorizado en el estilo. La primera de las etiquetas sigue teniendo letras grandes porque hemos dejado la asignación `FontSize = "20"` en su definición. Cuando usted pone explícitamente un valor en la definición de un elemento, este valor prevalece sobre el valor que se indicado en un estilo y será entonces el que aplique al elemento.

Los estilos tienen un atributo `TargetType` al que se le da como valor el tipo de elemento al que queremos se pueda aplicar el estilo. El estilo está compuesto por elementos `Setter`. Un elemento `Setter` tiene un atributo `Property` cuyo valor es el nombre de la propiedad en el `TargetType` al que se le va a cambiar el valor, y un atributo `Value` cuyo valor es el que se le va a aplicar a la propiedad indicada en `Property`. En el ejemplo del Listado 11-3 el primer `Setter` indica que la propiedad `FontFamily` del tipo `Label` debe tomar el valor "Candara". Del mismo modo hemos añadido un `Setter` por cada propiedad que se desea configurar de manera uniforme en todas las etiquetas.

Al tener estas características comunes a las etiquetas reunidas en un estilo, cambiar luego una característica se reduce a modificar el `Setter` correspondiente. En el ejemplo en que queríamos cambiar la fuente de letra basta con variar el `Setter` de la propiedad `FontSize` y darle un valor distinto (17 por ejemplo), el resultado puede verse en el Listado 11-4 y la Figura 11-3.

Listado 11-4 Cambio de tamaño de letra

```

<Window x:Class="StylesIntro.Window1" ...
       Title="Cambio de Contraseña" Height="399" Width="445">
    <Window.Resources>
        <Style TargetType="{x:Type Label}">
            <Setter Property="FontFamily" Value="Candara"/>
            <Setter Property="FontSize" Value="17"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Top"/>
            <Setter Property="Margin" Value="20,5"/>
        </Style>
    </Window.Resources>

```

```

</Window.Resources>
<Grid>
...
<Label Grid.ColumnSpan="2">
    Cambio de contraseña
</Label>
<Label Grid.Row="1">
    Contraseña anterior:
</Label>
<Label Grid.Row="2">
    Nueva contraseña:
</Label>
<Label Grid.Row="3">
    Confirmar contraseña:
</Label>
...
</Grid>
</Window>

```

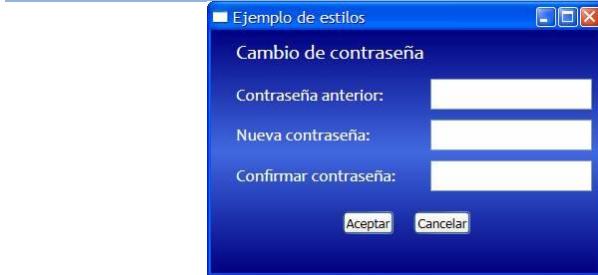


Figura 11-3 Ventana similar a la de la Figura 11-2 pero con las letras más grandes.

Los estilos que se pongan en los recursos de un elemento en WPF se aplican a todos los elementos que sean del tipo indicado en la propiedad `TargetType` del estilo y que estén contenidos en cualquier nivel de profundidad a partir del elemento donde se ha definido el estilo. En este ejemplo, hemos puesto el estilo en los recursos de la ventana, lo cual provoca que el estilo se aplique a todos los `Labels` que estén dentro de la misma.

Si además de estos cambios se deseara por ejemplo, que las cajas de contraseña fueran más estrechas y que los dos botones fueran de un mismo tamaño (Figura 11-4), basta con agregar dos estilos en los recursos de la ventana, uno para cada tipo de elemento (`PasswordBox` y `Button`) como se muestra en el Listado 11-5 y Figura 11-4.

Listado 11-5 Inclusión de estilos para `PasswordBox` y `Button`

```

<Window x:Class="StylesIntro.Window1" ...
       Title="Cambio de Contraseña" Height="399"
       Width="445">
    <Window.Resources>
        <Style TargetType="{x:Type Label}">
            <Setter Property="FontFamily" Value="Candara"/>
            <Setter Property="FontSize" Value="17" />

```

```

<Setter Property="HorizontalAlignment" Value="Left"/>
<Setter Property="VerticalAlignment" Value="Top"/>
<Setter Property="Margin" Value="20,5"/>
</Style>
<Style TargetType="{x:Type PasswordBox}">
<Setter Property="Height" Value="23"/>
</Style>
<Style TargetType="{x:Type Button}">
<Setter Property="Width" Value="70"/>
<Setter Property="Height" Value="30"/>
</Style>
</Window.Resources>
<Grid>
...
</Grid>
</Window>

```

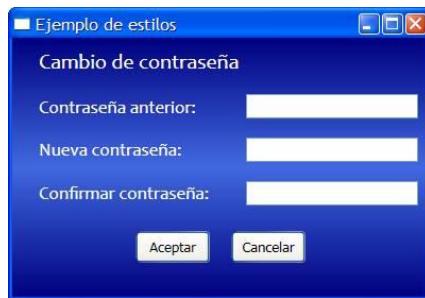


Figura 11-4 Ventana de la Figura 11-3 con cambios de estilo en las cajas de contraseñas y los botones.

11.3 Definición explícita de estilos

Los estilos en los ejemplos anteriores fueron aplicados implícitamente, es decir, los estilos que se ponen en los recursos se aplican automáticamente a todos los elementos del tipo que se indicó en el estilo y que estén en el mismo ámbito (o anidados en éste) en el que se ha puesto el recurso.

Sin embargo los estilos pueden ser aplicados también explícitamente porque podemos tener diferentes estilos para aplicar a un mismo tipo de elemento. Por ejemplo, en dependencia de nuestros intereses de la apariencia podemos querer que determinadas etiquetas aparezcan en un estilo y otras en otro. En este caso entonces no podemos aplicar implícitamente un estilo a todos los elementos. Para lograr esto la forma de diferenciar los estilos a un mismo destinatario (*TargetType*) es dándole un valor al atributo *Key* del estilo lo que indicará la llave con la que se guarda en los recursos.

En nuestro ejemplo queremos poner estilos diferentes a las etiquetas que sirven de encabezado a un diálogo de las etiquetas que sirven para nombrar cajas de texto. Note en el Listado 11-6 el

código XAML x:Key="Heading" y x:Key="Normal". Note como ahora es cada elemento quién decide cuál estilo se aplica

Listado 11-6 Diferentes estilos para un mismo tipo de elemento referidos explícitamente

```
<Window x:Class="StylesIntro.Window1" ...
    Title="Cambio de Contraseña" Height="399" Width="445">
<Window.Resources>
    <Style TargetType="{x:Type Label}" x:Key="Heading">
        <Setter Property="FontFamily" Value="Candara"/>
        <Setter Property="FontSize" Value="20"/>
        <Setter Property="HorizontalAlignment" Value="Left"/>
        <Setter Property="VerticalAlignment" Value="Top"/>
        <Setter Property="Margin" Value="20,5"/>
    </Style>
    <Style TargetType="{x:Type Label}" x:Key="Normal">
        <Setter Property="FontFamily" Value="Candara"/>
        <Setter Property="FontSize" Value="17"/>
        <Setter Property="HorizontalAlignment" Value="Left"/>
        <Setter Property="VerticalAlignment" Value="Top"/>
        <Setter Property="Margin" Value="20,5"/>
    </Style>
    ...
</Window.Resources>
<Grid>
    ...
    <Label Grid.ColumnSpan="2" Style="{StaticResource Heading}">
        Cambio de contraseña
    </Label>
    <Label Grid.Row="1" Style="{StaticResource Normal}">
        Contraseña anterior:
    </Label>
    <Label Grid.Row="2" Style="{StaticResource Normal}">
        Nueva contraseña:
    </Label>
    <Label Grid.Row="3" Style="{StaticResource Normal}">
        Confirmar contraseña:
    </Label>
    ...
</Grid>
</Window>
```

El resultado es idéntico al de la Figura 11-4. Note que ahora se le han puesto nombres diferentes a los dos estilos, de forma que cada etiqueta refiere a su estilo particular por el nombre usando la extensión de marca StaticResource.

11.4 Estilos basados en otros estilos

Los estilos nombrados que se mencionaron en la sección anterior son también útiles para definir estilos basados en otros estilos (algo así como "herencia de estilos"), note que en el XAML anterior los estilos Heading y Normal sólo se diferencian en el tamaño de la letra. Al igual que se logra en Microsoft Word podemos evitar estar repitiendo código similar en la definición de estilos si hacemos que unos estilos estén basados en otros. Así por ejemplo con la propiedad BasedOn de un elemento Style podemos hacer que el estilo Heading "herede" del estilo Normal todos sus elementos Setter y además pueda añadir nuevos Setter o redefinir algunos ya existentes en Normal. Esto se ilustra en el Listado 11-7.

Listado 11-7 Estilos basados en otros estilos
<pre><Window x:Class="StylesIntro.Window1" ... Title="Cambio de Contraseña" Height="399" Width="445"> <Window.Resources> <Style TargetType="{x:Type Label}" x:Key="Normal"> <Setter Property="FontFamily" Value="Candara"/> <Setter Property="FontSize" Value="16"/> <Setter Property="HorizontalAlignment" Value="Left"/> <Setter Property="VerticalAlignment" Value="Top"/> <Setter Property="Margin" Value="20,5"/> </Style> <Style TargetType="{x:Type Label}" x:Key="Heading"> BasedOn="{StaticResource Normal}" <Setter Property="FontSize" Value="20"/> </Style> ... </Window.Resources> <Grid> ... <Label Grid.ColumnSpan="2" Style="{StaticResource Heading}"> Cambio de contraseña </Label> <Label Grid.Row="1" Style="{StaticResource Normal}"> Contraseña anterior: </Label> <Label Grid.Row="2" Style="{StaticResource Normal}"> Nueva contraseña: </Label> <Label Grid.Row="3" Style="{StaticResource Normal}"> Confirmar contraseña: </Label> ... </Grid></pre>

```
</Window>
```

De esta manera (al igual que ocurre en Word) si ahora se cambia el tipo de letra del estilo Normal, el cambio se propaga al estilo Heading y los elementos con este estilo también cambiarían su tipo de letra.

Otra de las bondades de la herencia de estilos es que esta es aplicable a estilos de diferente tipo. Si por ejemplo usted tiene un estilo aplicable a elementos de tipo FrameworkElement, luego usted puede definir estilos específicos para cualquier elemento que herede de FrameworkElement que estén basados en este estilo inicial. En el Listado 11-8 y mostramos un estilo aplicable a FrameworkElement que pone un efecto de sombra a todos los elementos a los que se aplique, luego hemos hecho que el estilo Normal y el estilo de los PasswordBox estén basados en este estilo. Compare la Figura 11-4 y la Figura 11-5. Note la aplicación de la sombra a todos los elementos no sólo a las etiquetas, observe que la etiqueta que hace de encabezado tiene a su vez sombra y un tamaño mayor de letra por estar basada en el estilo Heading que está basado en Normal que a su vez está basado en Shadow.

Para los más conocedores de POO esto les puede parecer una suerte de "covarianza de estilos". Un estilo mas general (Shadow) se aplica a un elemento mas general (FrameworkElement) y a partir de este estilo se puede definir un estilo mas especializado (Normal) para aplicar a un tipo de elemento mas especializado del elemento general (Label)

Listado 11-8 Inclusión de estilo que aplica sombra a elementos de tipo FrameworkElement

```
<Window x:Class="StylesIntro.Window1" ...  
      Title="Cambio de Contraseña" Height="399" Width="445">  
<Window.Resources>  
  <Style TargetType="{x:Type FrameworkElement}" x:Key="Shadow">  
    <Setter Property="BitmapEffect">  
      <Setter.Value>  
        <DropShadowBitmapEffect ShadowDepth="1.4" Color="#8000"  
          Direction="315" Softness="0.2"/>  
      </Setter.Value>  
    </Setter>  
  </Style>  
  <Style TargetType="{x:Type Label}" x:Key="Normal"   
        BasedOn="{StaticResource Shadow}">  
    <Setter Property="FontFamily" Value="Candara"/>  
    <Setter Property="FontSize" Value="16"/>  
    <Setter Property="HorizontalAlignment" Value="Left"/>  
    <Setter Property="VerticalAlignment" Value="Top"/>  
    <Setter Property="Margin" Value="20,5"/>  
  </Style>  
  <Style TargetType="{x:Type Label}" x:Key="Heading"   
        BasedOn="{StaticResource Normal}">  
    <Setter Property="FontSize" Value="20"/>
```

```

</Style>
<Style TargetType="{x:Type PasswordBox}" BasedOn="{StaticResource
Shadow}">
    <Setter Property="Height" Value="23"/>
</Style>
...
</Window.Resources>
<Grid>
...
</Grid>
</Window>

```

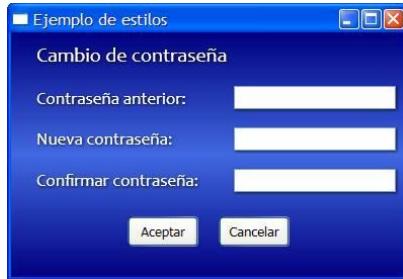


Figura 11-5 Label y PaswordBox con sombra

Usted además podría agregar otro estilo de definición implícita basado en Normal que no aplique ningún nuevo Setter ni cambio alguno en los anteriormente definidos. Le dejamos a usted encontrar la utilidad de un estilo como este.

Lección 12 Plantillas de Datos

Esta lección está dedicada a los controles que tienen un contenido a mostrar. Veremos cómo mediante una plantilla podemos agrupar características de apariencia y especificar cómo aplicarlas para mostrar los diferentes tipos de datos de una aplicación, de tal modo que esta plantilla pueda ser reutilizada en distintos lugares sin tener necesidad de replicar código XAML.

Al ejecutar el código XAML del Listado 12- 1, se despliega una ventana como la de la Figura 12- 1.

Listado 12- 1 Código para mostrar el contenido de un TextBlock

```

<Window x:Class="WPF_DataTemplates.FuentesConEstilo"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Nombres de Fuentes con Estilo" Height="124" Width="450">

```

```

<Grid DataContext="Times New Roman">
    <TextBlock      Text="{Binding}"      FontFamily="{Binding}"
    ToolTip="{Binding}"
        FontSize="18" >
        <TextBlock.BitmapEffect>
            <DropShadowBitmapEffect Softness="0.2" Opacity="0.7"
    ShadowDepth="2"/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</Grid>
</Window>

```



Figura 12- 1 Resultado de la ejecución del Listado 12-1

Note en el código la asignación al atributo `DataContext="Times New Roman"`. Como se ve en la Lección **Enlace a Datos** con esto se permite utilizar una fuente común de datos para todos los elementos contenidos en ese contexto (que en este ejemplo es el elemento `Grid`).

Note que en el código del Listado12-1 hay tres propiedades asociadas a esta misma fuente de datos

```
<TextBlock Text="{Binding}" FontFamily="{Binding}" ToolTip="{Binding}" ...>
```

En todos los casos se han enlazado esta propiedad con la cadena "Times New Roman" asociada como valor a `DataContext`. Al dar este valor a la propiedad `Text` este se asigna tal cual como la cadena ya que la propiedad `Text` de un `TextBlock` debe tener valores que sean cadenas. Sin embargo, al dar este mismo valor "Times New Roman" a la propiedad `FontFamily` es interpretado por WPF como la indicación de la tipología de caracteres de nombre **Times New Roman** (repase de la Lección **Enlace a Datos** que lo que está haciendo realmente WPF tras el telón es una conversión del dato del tipo de la fuente al tipo del destino). Es por ello que en la Figura 12- 1 el texto aparece en la tipología indicada, es decir Times New Roman. Y al asociar la propiedad `ToolTip` con la misma fuente de datos significará que cuando el cursor del ratón esté sobre el `TextBlock` se despliegue un *tool tip* con el texto contenido del `TextBlock` (la cadena "Times New Roman" en este caso). Note que dentro del *tool tip* la cadena aparece con la tipología propia del *tool tip* que no tiene nada que ver con la tipología **Times New Roman** que se le ha dado al `TextBlock`.

Estamos acostumbrados a ver traducido *font* por *fuente*. Para evitar confusión con lo que llamamos *fuente* de datos en el caso de un enlace, hemos utilizado aquí *tipología de caracteres* como traducción del término *font*..

Ud. puede pensar que usar un enlace a datos es demasiada parafernalia para un caso tan simple, que bien pudiésemos haber logrado escribiendo directamente en el código

```
<TextBlock Text="Times New Roman" FontFamily="Times New Roman"  
ToolTip="Times New Roman" ...>
```

Una justificación plausible de haber usado enlaces es que si durante la ejecución se cambiase el valor de la propiedad DataContext del Grid más externo, como se muestra en el Codebehind del Listado 12- 2, se obtendría un cambio como el que se observa en la Figura 12- 2.

Listado 12- 2 Codebehind para modificar la fuente de datos del Grid

```
...  
public FuenteConEstilo() {  
    InitializeComponent();  
    this.MouseUp += new  
        MouseButtonEventHandler(FuenteConEstilo_MouseUp);  
}  
void FueteConEstilo_MouseUp(object sender, MouseButtonEventArgs e) {  
    Grid grid = Content as Grid;  
    grid.DataContext = "Broadway";  
}  
...
```



Figura 12- 2 Resultado de cambiar la fuente de datos por un nuevo mensaje

Sin embargo, aún con esta solución de enlaces, si quisiéramos que otro TextBlock en la misma aplicación se tratase de la misma manera habría que replicar el código XAML, es decir escribir

```
<TextBlock Text="{Binding}" FontFamily="{Binding}" ToolTip="{Binding}" ...>
```

para dicho TextBlock.

Para facilitar la reutilización de un efecto como el anterior, sin tener que repetir código, WPF introduce el concepto de **plantilla (template)**. En esta lección veremos el tipo DataTemplate para definir plantillas que nos indiquen cómo desplegar los datos provenientes de una fuente que puede ser un objeto de negocio de la aplicación, un documento XML, un Servicios Web, etc.

Hay también plantillas para especificar la apariencia de controles, estas se estudian en la Lección **ControlTemplates**.

12.1 Plantilla de datos para aplicar a un contenido simple

Como primer ejemplo vamos a crear una plantilla para utilizar cuando se vayan a desplegar datos en controles cuyo contenido es (o puede ser) una cadenas de texto (por ejemplo un Label o un Button) y queremos que estos contenidos se desplieguen con una apariencia similar a la utilizada de la sección anterior.

Para aprovechar un código similar al que usamos en el Listado 12-1 vamos a definir un DataTemplate y para que sea reutilizable lo vamos a poner en los recursos de la ventana, como se muestra en el Listado 12-3.

Listado 12-3 Creación de una plantilla de datos para el tipo string

```
<Window ...>
<Window.Resources>
...
<DataTemplate x:Key="fontFamilyDT">
    <TextBlock Text="{Binding}" FontFamily="{Binding}" ToolTip="{Binding}"
        FontSize="18" >
        <TextBlock.BitmapEffect>
            <DropShadowBitmapEffect Softness="0.2" Opacity="0.7"
                ShadowDepth="2"/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</DataTemplate>
...
</Window.Resources>
...
</Window>
```

Este nuevo elemento DataTemplate agrupa los efectos que queríamos lograr al mostrar una cadena de texto (que si el nombre del texto coincidía con el nombre de una tipología el texto se escribiese con dicha tipología y que además apareciese un tool tip sobre el control al que se le asociase la plantilla). A este elemento le hemos asociado una llave (fontFamilyDT) y lo hemos puesto en los recursos de la ventana. Esto permite que podamos referirnos a esta plantilla desde otros lugares del código usando el nombre de la llave.

En esta plantilla también hemos incluido otro elemento de apariencia que es aplicar un efecto Bitmap al contenido del TextBlock, en este caso para que aparezca sombreado. Los efectos de Bitmap se estudian en la Lección **Efectos Visuales**.

Hay que mencionar que cada elemento definido en Listado 12-3, es tan solo una descripción del árbol que WPF debe construir al utilizar la plantilla en tiempo de ejecución. Es decir, a diferencia de los elementos que se describen fuera de la plantilla, que son creados inmediatamente al leer el documento XAML, en ejecución WPF al cargar el XAML no crea los elementos visuales correspondientes a los descritos dentro de la plantilla. Los elementos que se definen dentro de la plantilla se almacenan en memoria a manera de un FrameworkElementFactory, y quedan listos para ser creados como objetos visuales reales en cada lugar en que se aplique la plantilla.

Ahora lo que resta por hacer es asociar dicha plantilla a cada elemento cuyo contenido se desea se muestre usando la apariencia indicada por la plantilla. Para ello hay que asignarle la plantilla a la propiedad ContentTemplate del elemento, como se ha hecho en el Listado 12-4. Note que al hacer ContentTemplate="{StaticResource fontFamilyDT}" estamos indicando aquí asociarle la plantilla, que introdujimos con llave fontFamilyDT en los recursos, al contenido de cada uno de los elementos Label. Como la plantilla fue definida para mostrarse como un TextBlock esto exige que el contenido de la etiqueta deba poder mostrarse de esa manera, eso es posible en este caso porque como contenido de cada etiqueta se ha puesto una cadena.

Listado 12-4 Vinculación de los controles de contenido con las plantillas de datos

```
<Window ...>
...
<StackPanel HorizontalAlignment="Left">
    <Label ContentTemplate="{StaticResource fontFamilyDT}">Arial</Label>
    <Label ContentTemplate="{StaticResource fontFamilyDT}">Times New
        Roman</Label>
    <Label ContentTemplate="{StaticResource fontFamilyDT}">Broadway</Label>
</StackPanel>
</Window>
```

El resultado de ejecutar este fragmento de código es lo que se muestra en la Figura 12-3 y si se pasa el ratón sobre alguna de ella se desplegará también el tool tip correspondiente



Figura 12-3 Visualización de tres etiquetas con una plantilla.

Aunque estamos acostumbrados a ver que el contenido de una etiqueta sea un texto, realmente un Label es un ContentControl es decir un control que puede tener como contenido cualquier

objeto. Si cambiamos el contenido de la primera de las etiquetas en el ejemplo del Listado 12-3 y escribimos

```
<Label ContentTemplate="{StaticResource fontFamilyDT}">  
    <Image Source="Windito Vistoso.gif"/>  
</Label>
```

entonces al intentar aplicar la plantilla WPF encuentra que no puede convertir la imagen a un texto para asignarla a la propiedad Text de la plantilla y por tanto da como resultado lo que se muestra en la Figura 12-4.

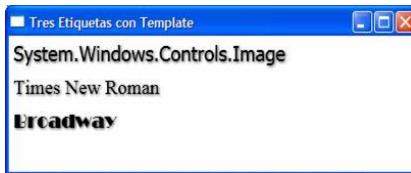


Figura 12-4 Incorrecta aplicación de una plantilla

En la Lección **Estilos** se estudia cómo obtener una apariencia uniforme para un conjunto de elementos del mismo tipo. De hecho una plantilla puede indicar la aplicación de un determinado estilo. Sin embargo, con las plantillas de datos podemos crear nuevas formas visuales orientadas a un tipo de datos o según el valor de un dato como veremos en la sección siguiente.

12.2 Plantilla para aplicar a un contenido no simple

En lugar de trabajar con un contenido simple como ha sido el caso de string utilizado en la sección anterior, usaremos los tipos Director y Film que han sido definidos en la Lección **Enlace a Datos**. La clase Film tiene las propiedades Título, Director, Género, Oscar, Calificación y Actores, y la clase Director tiene Nombre, Nacionalidad y Filmes. Las propiedades Actores y Filmes son de tipo colección.

Queremos definir una plantilla para lograr que los datos de un Film se muestren como se ilustra en la Figura 12-5. Para definir una plantilla hay que saber cómo son los tipos de datos que se van a representar mediante ésta. En este caso para simplificar hemos dejado que algunos datos de un film como son la lista de Actores, el Nombre del director y el Titulo de la película se representen en la forma predeterminada que asume WPF (solo hemos incluido algunos cambios de color o tamaño y tipo de fuente). Pero otros datos del film como es la Calificación que se le ha dado, la Nacionalidad del director y la indicación de si la película ganó Oscar los vamos a presentar usando plantillas.



Figura 12-5 Visor de Film usando DataTemplate.

Como muestra el Listado 12-5, el único contenido directo de la ventana es un ContentControl que significa un elemento que tiene un contenido, el cual es manejable a través de la propiedad Content. A esta propiedad Content de este elemento se le asigna instancia de Film (en este caso la instancia correspondiente al film JurassikPark).

Por brevedad en este ejemplo estamos asumiendo que existe una clase DatosFilmes con una propiedad o variable estática JurassikPark que nos da la instancia de film que queremos mostrar aplicándole la plantilla.

A la propiedad ContentTemplate se le ha asignado la plantilla plantillaFilmDatos que debemos haber ubicado en los recursos con valor de llave plantillaFilmDatos y que ha sido diseñada para visualizar los datos de un Film en la forma mostrada en la Figura 12-5.

Listado 12-5 Ventana para el visor de film
<pre><Window x:Class="WPF_DataTemplates.VisorPelicula" ... xmlns:ejemplo="clr-namespace:WPF_DataTemplates" Title="Cartelera de Cine" Height="250" Width="300" Background="#FF394472" Foreground="White" > <ContentControl Content="{x:Static ejemplo:DatosFilmes.JurassikPark}" ContentTemplate="{StaticResource plantillaFilmDatos}" Margin="6"/> </Window></pre>

En el Listado 12-6 se muestra la definición de la plantilla plantillaFilmDatos diseñada para visualizar al tipo de datos Film.

Listado 12-6 Plantilla de datos para mostrar un Film
<pre><DataTemplate x:Key="plantillaFilmDatos"> <Grid> <Grid.RowDefinitions> <RowDefinition Height="Auto"/> <RowDefinition Height="Auto"/> <RowDefinition/></pre>

```

</Grid.RowDefinitions>
<ContentControl Content="{Binding Titulo}"
    ContentTemplate="{StaticResource plantillaTítulo}"/>
<ContentControl Grid.Row="1" Content="{Binding Oscar}"
    ContentTemplate="{StaticResource plantillaOscar}" Margin="6"
    HorizontalAlignment="Right" VerticalAlignment="Top"/>
<ContentControl Grid.Row="1" Content="{Binding Director.Nacionalidad}"
    ContentTemplate="{StaticResource plantillaNacionalidad}"
    Margin="6" HorizontalAlignment="Left"
    VerticalAlignment="Top"/>
<ContentControl Grid.Row="1" Content="{Binding Calificación}"
    ContentTemplate="{StaticResource plantillaRating}" Margin="6"
    HorizontalAlignment="Center" VerticalAlignment="Top"/>
<StackPanel Grid.Row="2">
    <TextBlock>
        Director:
        <TextBlock Text="{Binding Director.Nombre}"/>
    </TextBlock>
    <TextBlock>Protagonistas:</TextBlock>
    <ItemsControl ItemsSource="{Binding Actores}" Margin="12 0 0 0"/>
</StackPanel>
</Grid>
</DataTemplate>

```

Note que se utilizan otros cuatro ContentControl que hemos puesto en la primera fila del Grid. En cada uno de estos ContentControl se visualizan diferentes datos de un film usando una plantilla específica para cada uno. Recuerde que al diseñar una plantilla de datos, debe asumir también que en el contexto de datos de todos los elementos dentro de la plantilla aparece el dato que se debe visualizar en cuestión, a menos se cambie la propiedad DataContext de uno de ellos. Por ejemplo, en el primer ContentControl, se enlaza la propiedad Content={Binding Titulo}, y como una instancia de Film es la que debe aparecer en el contexto entonces se le puede pedir la propiedad Titulo.

También es posible utilizar rutas complejas dentro de una plantilla, como la que usamos en el tercer ContentControl del Listado 12-6 cuando hacemos

Content={Binding Director.Nacionalidad} que nos permite acceder a la nacionalidad del director del film. A este valor nacionalidad del director se le aplicará la plantilla plantillaNacionalidad que nos muestra la bandera del país. La definición de esta plantilla se muestra en el Listado 12-7.

Listado 12-7 Descripción de una plantilla para mostrar la imagen de una bandera por cada nacionalidad

```

<ejemplo:FlagConverter x:Key="flagConverter" />
<DataTemplate x:Key="plantillaNacionalidad">
    <Image Width="30" Height="20" Stretch="Uniform"
        Source="{Binding Converter={StaticResource flagConverter}}"
        Margin="2"
        ToolTip="{Binding}"/>
</DataTemplate>

```

Observe que la plantilla solo contiene una imagen y las propiedades Source (de tipo `ImageSource`) y `ToolTip` (de tipo `string`). Como `Source` no corresponde con el tipo de datos en la fuente del enlace, es necesario utilizar un conversor que nos indique cómo convertir una cadena en una imagen (en este caso la imagen de la bandera). Para esto en el code-behind se ha definido la clase `FlagConverter` que se muestra en el Listado 12-8.

Listado 12-8 Conversor de valores de tipo string a imágenes extraídas de los recursos

```

public class FlagConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
        CultureInfo cultureInfo){
        if (!(value is string)) return null;
        string uriSource = "/imagenes/" + (value as string) + ".bmp";
        return new BitmapImage(new Uri(uriSource, UriKind.Relative));
    }
    public object ConvertBack(object value, Type targetType, object
        parameter,
        CultureInfo cultureInfo) {
        return null;
    }
}

```

En la carpeta imágenes del proyecto hemos ubicado varias imágenes una por cada nacionalidad de manera que al convertir el valor "USA", por ejemplo, el conversor pueda encontrar un recurso de imagen asociado al Uri "/imagenes/USA.bmp". En la Figura 12-6 se muestra el explorador de solución en Visual Studio que nos muestra cómo configurar el proyecto para colocar las imágenes dentro de los recursos. En el panel de propiedades del recurso se observa cómo se marca la opción `Resource`. El objeto `BitmapImage` que se retorna en el método `Convert` es un heredero de `ImageSource` que construye un objeto imagen a partir de un archivo.

12.3 Plantillas a Colecciones de Elementos

WPF al visualizar un control que tenga un contenido múltiple para la presentación de una colección lo que hace es iterar sobre la colección y aplicar una plantilla a cada elemento de la

colección (puede que una plantilla diferente por elemento), desplegando la representación visual resultante.

Los controles de contenido múltiple heredan de ItemsControl que tiene una propiedad Items de tipo ItemCollection. Los conocidos ListBox, ComboBox,ToolBar, MenuItem, Menu, TabControl, etc son descendientes de ItemsControl. El Listado 12-9 nos muestra un ListBox con varios elementos de diferente tipo (un objeto ListBoxItem, y varios botones y un objeto Film) Los elementos se añaden a la colección de la propiedad Items.

Este código despliega lo que se muestra en la Figura 12-10. A falta de una plantilla para mostrar cada elemento WPF muestra cada uno en la forma predeterminada. Observe que en el caso del último elemento de la colección que es un objeto de tipo Film se ha mostrado solo el título, esto se debe a que en este caso de forma predeterminada WPF muestra el string devuelto por la aplicación del método ToString de Film el cual hemos definido para que devuelva el valor de la propiedad Titulo.

Listado 12-9 Declaración explícita en XAML de elementos dentro de un ListBox

```
<Window ...>
...
<ListBox Margin="4">
    <ListBoxItem>Seleccionar todos</ListBoxItem>
    <Button>Infiltrados</Button>
    <Button>Goodfellas</Button>
    <Button>Taxi Driver</Button>
    <Button>Casablanca</Button>
    <ejemplo:Film Titulo="La Lista de Schindler" Oscar="true"/>
    <ejemplo:Film Titulo="Conociendo a los Focker" Oscar="false"/>
    <ejemplo:Film Titulo="Los Puentes de Madison" Oscar="false"/>
    <ejemplo:Film Titulo="Bailando con Lobos" Oscar="true"/>
</ListBox>
</Window ...>
```

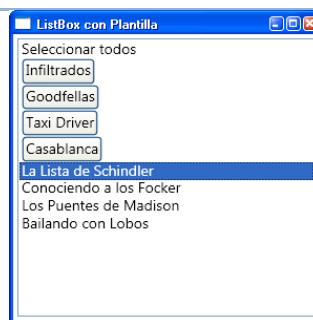


Figura 12-7 ListBox con diferentes tipos de objetos

Con el propósito de utilizar enlace de datos desde una fuente que represente una colección, se utiliza la propiedad `ItemsSource` que es de tipo `IEnumerable`. Al asignar o enlazar un valor a esta propiedad, el control se hace cargo de la fuente de datos, itera la colección, verifica si la instancia de colección implementa la interfaz `INotifyCollectionChanged` para anotarse al evento de cambio de colección, y listo. Luego de estas operaciones de inicialización, el control obtiene a través de la propiedad `Items` los elementos que debe visualizar.

En este curso no se hace hincapié en la interfaz `INotifyCollectionChanged` ya que la clase genérica `ObservableCollection<T>` (espacio de nombres `System.Collections.ObjectModel` en el ensamblado `WindowsBase`) implementa la interfaz. Basta con heredar de esta clase asignando al parámetro genérico `T` algún tipo apropiado a nuestros fines, y redefinir algunos de los métodos `InsertItem`, `RemoveItem`, `MoveItems`, `ClearItems` o `SetItem`, si es necesario hacerlo.

12.3.1 Aplicando plantillas a los elementos de una colección

Un control de contenido múltiple (`ItemsControl`) permite el uso de plantillas para aplicar a los elementos de la colección.

Mediante la propiedad `ItemTemplate` es posible asociar la misma plantilla de datos a todos los elementos del control de contenido múltiple a la vez que es posible asociar individualmente a cada elemento una plantilla y lograr así el máximo nivel de granularidad en el uso de plantillas de datos. En el Listado 12-10 se tiene un primer elemento de tipo `ListBoxItem` que tiene su propia plantilla `stringDT` (por brevedad no se ha incluido aquí la definición de la misma) y por tanto no se le aplicará a éste la plantilla general que se le ha indicado a `ItemTemplate` (sección en negrita dentro del Listado 12-10). A los restantes elementos sí se les asocia esta plantilla general que ha sido definida dentro del propio `ListBox`. Esta plantilla hace que cada item se despliegue como un `CheckBox` el cual además enlaza la propiedad `IsChecked` con la propiedad `Oscar` del objeto fuente de datos. Ver resultado en la Figura 12-8

Listado 12-10 Uso de plantillas individuales en un `ListBox` mediante elementos de tipo `ListBoxItem`

```
<Window ...>
...
<ListBox Margin="4" FontSize="14">
    <ListBoxItem ContentTemplate ="{StaticResource stringDT}">
        Seleccionar todos
    </ListBoxItem>
    <!--<ListBoxItem>Seleccionar todos</ListBoxItem>-->
    <Button>Infiltrados</Button>
    <Button>Goodfellas</Button>
    <Button>Taxi Driver</Button>
    <Button>Casablanca</Button>
    <example:Film Titulo="La Lista de Schindler" Oscar="true"/>
```

```

<ejemplo:Film Titulo="Conociendo a los Focker" Oscar="false"/>
<ejemplo:Film Titulo="Los Puentes de Madison" Oscar="false"/>
<ejemplo:Film Titulo="Bailando con Lobos" Oscar="true"/>
<ListBox.ItemTemplate>
    <DataTemplate>
        <CheckBox Content="{Binding}" IsChecked="{Binding Oscar}"></CheckBox>
    </DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</Window ...>

```



Figura 12-8 Uso de una plantilla individual y una colectiva en un ListBox

Observe en este ejemplo que solamente a los últimos elementos del ListBox (que se han incluido dentro del ListBox como objetos de tipo Film) se les puede activar correctamente el enlace, porque son los únicos elementos que pueden hacer una correspondencia con la ruta Oscar especificada en el enlace de la plantilla. A los demás elementos, que son de tipo Button, también se les aplica la plantilla (note que por eso han sido desplegados también con el pequeño cuadrado de marca de todo CheckBox). En este caso como estos botones no tienen una propiedad Oscar no se enlaza ningún valor con la propiedad IsChecked y el CheckBox de los botones por eso sale sin la marca (Figura 12-8).

Observe que la plantilla se ha definido directamente dentro del elemento ListBox.ItemTemplate, esta es otra forma de hacerlo, en lugar de poner a la plantilla en los recursos pero entonces claro está solo podrá aplicarse la plantilla dentro del ámbito del ListBox.

12.3.2 Patrón Maestro Detalle para mostrar colecciones y sus elementos

Para ilustrar el uso de plantilla en controles de contenido múltiple, vamos a mostrar una cartelera de films.

Un patrón utilizado para no mostrar todos los detalles de cada elemento de una colección es el conocido como **Maestro-Detalle**. Se muestra la colección con un extracto de la información por

cada elemento (**Maestro**) y paralelamente se muestran los detalles (**Detalle**) del elemento que se haya seleccionado dentro de la colección.

En el Listado 12-11 se utiliza este patrón para visualizar una lista de directores de cine, por cada uno de los cuales se muestra la lista de películas disponibles, y para cada película que se seleccione de un director se muestran detalles de la misma.

Listado 12-11 Uso del patrón maestro detalle y plantillas para crear una cartelera decine

```
<Window x:Class="WPF_DataTemplates.CarteleraCine"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ejemplo="clr-namespace:WPF_DataTemplates"
    Title="Cartelera de Cine" Height="450" Width="500"
    Background="#FF394472" >
<Window.Resources>
    ...
</Window.Resources>
<Grid TextBlock.Foreground="White" TextBlock.FontSize="14"
    TextBlock.FontFamily="Cambria, Book Antiqua, Times New Roman,
serif">
    ...
<DockPanel>
    <ContentControl DockPanel.Dock="Top" Margin="6"
        ContentTemplate="{StaticResource plantillaTítulo}">
        Directores</ContentControl>
    <ListBox DockPanel.Dock="Top" Name="lbDirectores"
        ItemsSource="{x:Static ejemplo:DatosFilmes.Directores}"
        ItemTemplate="{StaticResource     plantillaDirectorItem}"
        Margin="6"/>
    </DockPanel>
    <DockPanel Grid.Column="1">
        <ContentControl DockPanel.Dock="Top" Margin="6"
            ContentTemplate="{StaticResource plantillaTítulo}">
            Filmes</ContentControl>
        <ListBox DockPanel.Dock="Top" Name="lbFilmes"
            ItemsSource="{Binding           SelectedItem.Filmes,
ElementName=lbDirectores}"
            ItemTemplate="{StaticResource plantillaFilmItem}" Height="150"
            Margin="6"/>
        <ContentControl     Content="{Binding           SelectedItem,
ElementName=lbFilmes}"
            ContentTemplate="{StaticResource plantillaFilmDatos}">
```

```

        Margin="6"/>
    
```

Observe que hemos reutilizado la plantilla del Listado 12-6 para mostrar los detalles del film seleccionado. Esto es posible porque la plantilla la definimos para contenidos que fuesen de tipo Film y el resultado que se asocia al Content del último ContentControl en el Listado 12-17 es el que se obtiene del enlace {Binding SelectedItem, ElementName=lbFilmes} que indica el film seleccionado y que es precisamente de tipo Film.

En el Listado 12-12 se muestran las plantillas que utilizamos para la lista de directores y la lista de filmes.

Listado 12-12 Asociación de plantillas a los elementos individuales de un ListBox

```

<DataTemplate x:Key="plantillaDirectorItem">
    <StackPanel Orientation="Horizontal">
        <ContentControl Content="{Binding Nacionalidad}"
            ContentTemplate="{StaticResource plantillaNacionalidad}" />
        <TextBlock VerticalAlignment="Center">
            <TextBlock Text="{Binding Nombre}" FontWeight="Normal"
                FontStyle="Italic"/>
            (<TextBlock Text="{Binding Filmes.Count}" />
            filmes)
        </TextBlock>
    </StackPanel>
</DataTemplate>
...
<DataTemplate x:Key="plantillaFilmItem">
    <StackPanel Orientation="Horizontal">
        <ContentControl Content="{Binding Calificación}"
            ContentTemplate="{StaticResource plantillaRating}"
            VerticalAlignment="Center" Margin="0 0 6 0">
            <ContentControl.LayoutTransform>
                <ScaleTransform ScaleX="0.5" ScaleY="0.5" />
            </ContentControl.LayoutTransform>
        </ContentControl>
        <ContentControl Content="{Binding Oscar}"
            ContentTemplate="{StaticResource plantillaOscar}" />
    <ContentControl.LayoutTransform>

```

```

<ScaleTransform ScaleX="0.2" ScaleY="0.2"/>
</ContentControl.LayoutTransform>
</ContentControl>
<TextBlock VerticalAlignment="Center" Margin="6 0 12 0">
    <TextBlock     Text="{Binding     Titulo}"     FontWeight="Normal"
FontStyle="Italic"/>
</TextBlock>
</StackPanel>
</DataTemplate>

```

El resultado final se muestra en la Figura 12-9.



Figura 12-9 Uso de plantillas en controles de contenido múltiple

12.3.3 DataTemplateSelector

La limitación con este enfoque de asociar *contenido - plantilla* (Content-ContentTemplate, Header-HeaderTemplate, Items-ItemTemplate) es que esta relación hay que expresarla explícitamente en el código y esto se hace por tanto estáticamente cuando se escribe el código. Sin embargo, podría ser deseable que en ocasiones la plantilla que se asocie a un contenido se decida en ejecución.

Para dar más flexibilidad y dinamismo WPF incluye una tercera propiedad cuyo nombre según el control es ContentTemplateSelector, HeaderTemplateSelector, ItemTemplateSelector. Todas estas propiedades (con sufijo TemplateSelector en su nombre) son de tipo DataTemplateSelector, un tipo que tiene un único método (Listado 12-13) que permite seleccionar la plantilla a aplicar.

Listado 12-13 Clase DataTemplateSelector,

```

public class DataTemplateSelector {
    public virtual DataTemplate SelectTemplate(object item,
                                                DependencyObject container) {
        return null;
    }
}

```

Si cuando WPF va a aplicar una plantilla el valor de ContentTemplate (HeaderTemplate, ItemTemplate) es null y el valor de ContentTemplateSelector (HeaderTemplateSelector, ItemTemplateSelector) es distinto de null, entonces se aplica el método SelectTemplate del objeto DataTemplateSelector que esté como valor de dicha propiedad ContentTemplateSelector y se aplica la plantilla que devuelva este método.

El parámetro item del método SelectTemplate recibe el contenido que se va a mostrar y al cual se le va a aplicar una plantilla. Por lo general la implementación del método SelectTemplate tiene en cuenta el tipo dinámico de este objeto item, o alguna condición impuesta a las propiedades de este contenido, para determinar la plantilla que usará.

El segundo parámetro DependencyObject container representa al elemento visual que contiene los elementos generados a partir de la plantilla. Este parámetro es el que permite por ejemplo cargar una plantilla de los recursos. El método FindResource nos servirá con este propósito si el parámetro es de tipo FrameworkElement.

Para el mismo ejemplo de cartelera de filmes, vamos a crear un selector de plantilla de datos (DataTemplateSelector) que nos permita decidir cuál plantilla utilizar para mostrar los filmes según su calificación. Utilizaremos la plantilla plantillaFilmMalItem para los filmes con Calificacion menor que 3 y la plantilla plantillaFilmItem, que ya definimos anteriormente, para el resto de los filmes. Un caso particular es cuando el parámetro item es null, esto ocurre por ejemplo cuando la ventana acaba de aparecer y no se ha seleccionado ningún director aún. Para este caso utilizaremos el valor null como valor de plantilla, que WPF lo trata como no mostrar nada.

La clase FilmTemplateSelector se implementa como aparece en el Listado 12-14.

Listado 12-14 implementación de un selector de plantillas de datos

```

public class FilmTemplateSelector : DataTemplateSelector {
    public override DataTemplate SelectTemplate(object item,
                                                DependencyObject container) {
        Film film = item as Film;
        FrameworkElement elem = container as FrameworkElement;
        if (elem == null) return null;
    }
}

```

```

        string templateName = null;
        if (film != null)
            if (film.Calificación < 3)
                templateName = "plantillaFilmMalItem";
            else
                templateName = "plantillaFilmItem";
            if (templateName == null) return null;
            return elem.FindResource(templateName) as DataTemplate;
        }
    }
}

```

Para poder utilizar este selector en el código XAML hay que declarar entonces un objeto de tipo FilmTemplateSelector en los recursos y asignarlo a la propiedad ItemTemplateSelector del control ListBox que muestra la lista de filmes. El Listado 12-15 muestra el fragmento de código XAML para lograr esto. Observe que ya no usamos el atributo ItemTemplate (de estar definidos ambos se le daría prioridad a éste sobre el que hubiese en ItemTemplateSelector).

Listado 12-15 Asignando un selector de plantilla a un control de contenido simple

```

<Window.Resources>
...
<ejemplo:FilmTemplateSelector x:Key="filmSelector"/>
...
</Window.Resources>
...
<DockPanel Grid.Column="1">
    <TextBlock DockPanel.Dock="Top" Margin="6">Filmes</TextBlock>
    <ListBox DockPanel.Dock="Top" Name="lbFilmes"
        ItemsSource="{Binding SelectedItem.Filmes,
ElementName=lbDirectores}"
        ItemTemplateSelector="{StaticResource filmSelector}"
        Height="150" Margin="6"/>
        <ContentControl Content="{Binding SelectedItem,
ElementName=lbFilmes}"
            ContentTemplate="{StaticResource plantillaFilmDatos}"
Margin="6"/>
</DockPanel>...

```

Observe en la Figura 12-10 los filmes que se muestran con una letra rojiza para indicar que no están calificados o tienen una calificación muy baja.



Figura 12-10 Cuando un film tiene una calificación baja, se selecciona una plantilla diferente

12.3.4 ItemsPanelTemplate

Los controles de contenido múltiple brindan también una manera de configurar mediante plantillas la apariencia de cómo se distribuyen los elementos dentro del control (su layout).

Por ejemplo, en el Listado 12-16 se muestra un menú con un submenú. Los elementos `MenuItem` y `MenuItem` heredan de `ItemsControl` y como muestra la Figura 12-11, ambos distribuyen los elementos de manera diferente a la hora de mostrarlos. `MenuItem` los distribuye en forma horizontal y `MenuItem` de forma vertical).

Listado 12-16 Elemento MenuItem con un submenu

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="File"/>
    <MenuItem Header="Edit"/>
    <MenuItem Header="Windows"/>
    <MenuItem Header="Help">
        <MenuItem Header="Contents"/>
        <MenuItem Header="Index"/>
        <MenuItem Header="Search"/>
        <MenuItem Header="About"/>
    </MenuItem>
</Menu>
```

Figura 12-11 La clase Menu distribuye los elementos de manera horizontal y MenuItem de manera vertical

Como se estudia en la Lección **Layout** los paneles son elementos visuales que pueden contener elementos de interfaz de cualquier tipo (su propiedad Children es una colección de los UIElement ubicados dentro del panel). Un panel se encarga de distribuir en su superficie los elementos contenidos en él. Al igual que la clase ItemsTemplate nos permite definir una plantilla para un elemento de una colección, la clase ItemsPanelTemplate nos permite describir una plantilla para expresar cómo distribuir los elementos dentro del control. Para ello todo elemento de tipo ItemsControl tiene también una propiedad ItemsPanel a la que puede dársele como valor un ItemsPanelTemplate.

Gracias a esta capacidad podemos indicarle a un control una plantilla que defina cómo distribuir sus elementos. En este ejemplo vamos asociar una plantilla a un MenuItem para que distribuya sus elementos de manera vertical (Listado 12-17) y hacer que estos se muestren como en la Figura 12-12. En este caso le hemos dado un StackPanel como plantilla a la propiedad ItemsPanel del menú. Recuerde de la Lección **Layout** que un StackPanel distribuye por defecto los elementos de forma vertical.

Listado 12-17 Uso de StackPanel para lograr una distribución vertical en los elementos de un Menu

```
<Menu DockPanel.Dock="Top">
  <MenuItem.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel/>
    </ItemsPanelTemplate>
  </MenuItem.ItemsPanel>
  <MenuItem Header="File"/>
  <MenuItem Header="Edit"/>
  <MenuItem Header="Windows"/>
  <MenuItem Header="Help"/>
  <MenuItem Header="Contents"/>
  <MenuItem Header="Index"/>
  <MenuItem Header="Search"/>
  <MenuItem Header="About"/>
</MenuItem>
</Menu>
```

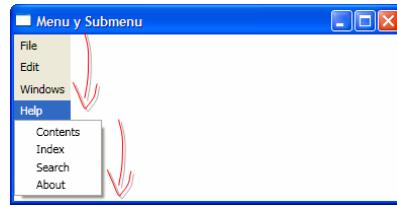


Figura 12-12 Menu con distribución de sus elementos de forma vertical

Para definir este tipo de plantilla a asociar a un `ItemPanel` solo se puede usar un panel (heredero de `Panel`) pero que no tenga elementos. Es decir un panel en el que solo se indique cuál y cómo vamos a usar su distribución para aplicarla a los elementos a los que se aplique la plantilla, es decir para aplicarla al valor de la propiedad `Items` del `ItemsControl`. Sin embargo a este panel sí se le pueden configurar sus propiedades. En el Listado 12-18 se le dan valores a las propiedades `Background` y `LayoutTransform` del panel con los que se le da color de fondo verde y un ángulo de rotación al menú (Figura 12-13).

Listado 12-18 Configuración del panel de una plantilla

```
<Menu Dock="Top" VerticalAlignment="Top">
    <HorizontalAlignment="Left">
        <MenuItem Header="File"/>
        <MenuItem Header="Edit"/>
        <MenuItem Header="Windows"/>
        <MenuItem Header="Help">
            <MenuItem Header="Contents"/>
            <MenuItem Header="Index"/>
            <MenuItem Header="Search"/>
            <MenuItem Header="About"/>
        </MenuItem>
    </Menu.ItemsPanel>
    <ItemsPanelTemplate>
        <StackPanel Background="LightGreen">
            <StackPanel.LayoutTransform>
                <RotateTransform Angle="45"/>
            </StackPanel.LayoutTransform>
        </StackPanel>
    </ItemsPanelTemplate>
</Menu>
```

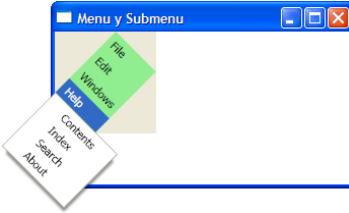


Figura 12-13 Afectación de la plantilla de distribución a la apariencia de un Menú

Observe en la Figura 12-13 el área gris que no aparece inclinada ni en verde, esa área corresponde al menú y no es lo mismo que el área del StackPanel que se ha asociado como plantilla a los elementos del menú.

En el Listado 12-19 se utiliza como plantilla del submenú un StackPanel al que se le ha indicado orientación horizontal para distribuir los elementos del submenú en sentido horizontal como se aprecia en la Figura 12-14.

Listado 12-19 Uso de la propiedad Orientation para dirigir el sentido de distribución en una plantilla

```
<Menu>
  <Menu.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel/>
    </ItemsPanelTemplate>
  </Menu.ItemsPanel>
  <MenuItem Header="File"/>
  <MenuItem Header="Edit"/>
  <MenuItem Header="Windows"/>
  <MenuItem Header="Help">
    <MenuItem.ItemsPanel>
      <ItemsPanelTemplate>
        <StackPanel Orientation="Horizontal"/>
      </ItemsPanelTemplate>
    </MenuItem.ItemsPanel>
    <MenuItem Header="Contents"/>
    <MenuItem Header="Index"/>
    <MenuItem Header="Search"/>
    <MenuItem Header="About"/>
  </MenuItem>
</Menu>
```



Figura 12-14 Menú vertical con submenú horizontal

Lección 13 Triggers

Con los estilos y plantillas podemos personalizar la visualización y apariencia de las aplicaciones, estos además facilitan la reutilización sin replicar código. Es decir, con estos recursos podemos definir el **QUÉ** y el **CÓMO** de un efecto o cambio en la apariencia de la interfaz de una aplicación.

Sin embargo, con los **triggers** (desencadenador) podemos especificar, declarativamente en la misma notación XAML, el **CUÁNDO** y el **POR QUÉ** aplicar efectos y modificaciones visuales (por lo general a través de plantillas y estilos) durante la ejecución de la aplicación.

Con un desencadenador podemos indicar si un estilo debe cambiar cuando ocurra algún evento (por ejemplo con el ratón o el teclado)

Un desencadenador representa por tanto una forma de “**desencadenar**” un conjunto de acciones, ya sea como respuesta al cambio de valor de una determinada propiedad, o a la ocurrencia de un determinado evento.

Aunque todos los elementos de tipo FrameworkElement tienen una propiedad Triggers, a esta solo puede asignarse un EventTrigger, los cuales veremos en esta Lección.

Antes de estudiar los tipos específicos de desencadenadores veamos primero algunas características generales comunes a todos ellos.

Los desencadenadores vienen a ser una forma de encapsular uno o más elementos Setter dentro de una suerte de condición, de manera que estos sean ejecutados si la condición es verdadera. Un desencadenador también tiene una propiedad SourceName que es usada cuando éste se define en una plantilla (template).

Un elemento Setter cuenta de dos atributos: Property para especificar el nombre de la propiedad que se modificará y Value que indica el valor que se le asigna a dicha propiedad. Ambos atributos deben especificarse o de lo contrario la ejecución del código provocará un error.

Si colocamos en el código XAML más de un desencadenador independientes unos de otros, y más de uno actúa sobre una misma propiedad (modificando su valor) el último valor es el que prevalece.

Los desencadenadores heredan de la clase base TriggerBase . Hay tres tipos fundamentales de desencadenadores: Trigger, DataTrigger y EventTrigger

13.1 Property Triggers

La forma más simple de desencadenador son los de tipo Trigger (llamados **Property Triggers**). Un Trigger chequea el valor de una determinada propiedad (**dependency property**) y en dependencia de este valor desencadena una determinada acción.

Deben especificarse valores para las propiedades Property y Value del Trigger. Los valores de estas propiedades Property y Value indican que el Trigger ejecutará lo que se ha indicado en sus Setter cuando Property tenga el valor especificado en Value.

En el Listado 13-1 Ud. puede apreciar el uso de un desencadenadores de tipo Trigger (destacado en negrita en el código). Con el que indicamos modificar el valor de la opacidad de la imagen (propiedad Opacity) cuando la propiedad IsMouseOver tenga valor True (es decir cuando el ratón pase por encima de la imagen).

Listado 13- 1 Código XAML, Trigger

```
<Window x:Class="Triggers.Window1"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Triggers: Property Trigger" Height="300" Width="300"
    Background="DarkBlue" >
    <Window.Resources>
        <Style TargetType="{x:Type Image}">
            <Setter Property="Image.Opacity" Value="0.5" />
        <Style.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="Image.Opacity" Value="1"/>
            </Trigger>
        </Style.Triggers>
    </Style>
    </Window.Resources>
    <Image Source="Windito.gif" Height="200"/>
</Window>
```

La Figura 13-1 nos muestra lo que ocurre cuando posicionamos el cursor del ratón sobre la imagen.



Figura 13 - 1 Modificando apariencia con un Trigger (antes y después).

Si la condición que provocó el desencadenamiento de Trigger deja de cumplirse, entonces las propiedades modificadas por el Trigger son automáticamente modificadas a su valor anterior (una suerte de **undo**). El sistema de dependency properties de WPF vela por la propiedad del Trigger para realizar este proceso inverso. Si en el ejemplo el cursor vuelve a salir fuera de la imagen el valor de la opacidad de la misma vuelve a ser el anterior.

Cuando un desencadenador Trigger se define dentro de un estilo (como ha sido el caso de este ejemplo), actuará siempre sobre el control al que se esté aplicando el estilo o la plantilla, sin embargo se puede aplicar a otros elementos visuales específicos contenidos en la plantilla asignando a la propiedad SourceName del Trigger el nombre del elemento específico al que se desea aplicar.

En algunos casos puede resultar interesante o necesario que un Trigger se ejecute cuando se cumpla más de una condición. Con un MultiTrigger puede especificarse un desencadenador que se aplique cuando se cumplan varias condiciones. . Observe a continuación un ejemplo de MultiTrigger (Listado 13-2) con más de una condición.

Listado 13- 2 Código XAML, MultiTrigger

```
<Window x:Class="Triggers.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Triggers: MultiTrigger" Height="300" Width="300"
    Background="DarkBlue" >
<Window.Resources>
    <Style TargetType="{x:Type Button}">
        <Style.Triggers>
            <MultiTrigger>
                <MultiTrigger.Conditions>
                    <Condition Property="IsMouseOver" Value="True"/>
                    <Condition Property="IsEnabled" Value="True"/>
                </MultiTrigger.Conditions>
                <Setter Property="BitmapEffect">
```

```

<Setter.Value>
    <OuterGlowBitmapEffect GlowColor="Gold" GlowSize="20"/>
</Setter.Value>
</Setter>
</MultiTrigger>
<Style.Triggers>
</Style>
</Window.Resources>
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Button Width="50" Height="50" FontFamily="Webdings"
        FontSize="20" FontWeight="Bold" IsEnabled="False"> 9 </Button>
    <Button Width="50" Height="50" FontFamily="Webdings"
        FontSize="20" FontWeight="Bold"> : </Button>
</StackPanel>
</Window>

```

Podemos combinar condiciones con la propiedad Conditions del MultiTrigger indicando dentro de ésta elementos de tipo Condition donde cada uno de estos indica una propiedad Property y un valor Value.

Observe en la Figura 13-2 el resultado de la ejecución del código XAML del Listado 13-2. Se muestran dos botones, uno deshabilitado y el otro no. Al estar el cursor del ratón sobre el que está habilitado se cumplirán las condiciones del MultiTrigger y se ejecutará lo que indica su Setter. Para el caso del que se encuentra deshabilitado las condiciones del MultiTrigger no se cumplen todas y por tanto no se ejecuta. Las propiedades modificadas volverán a su estado inicial una vez que deje de cumplirse una de las condiciones del MultiTrigger.



Figura 13 - 2 Modificando apariencia con un MultiTrigger.

En este ejemplo estamos cambiando la apariencia del botón agregando el efecto de bitmap OuterGlowBitmapEffect al botón, provocando una suerte de iluminación alrededor del mismo. Existen otros efectos de bitmap en WPF que pueden aplicarse a un UIElement estos se estudian en la Lección **Efectos Visuales**.

13.2 DataTriggers

Un DataTrigger es similar a un Trigger excepto que para expresar la condición no se usa una propiedad del propio elemento sobre el que actuará el desencadenador sino que se usa un Binding para poder preguntar por propiedades del elemento al que nos enlazamos con el Binding.

A diferencia de los Trigger que chequean valores de dependency property, un DataTrigger pueden preguntar por el valor de cualquier objeto .NET representado en una propiedad.

Considere las clases del Listado 13-3 para llevar una estadística de resultados de fútbol en lo que va de temporada. El código XAML del Listado 13-4 tiene un DataTrigger que se basa en una condición que lo que hace es preguntar por el valor de la propiedad Ganador definida en el Codebehind del Listado 13-3.

Listado 13-3 Implementación en el Codebehind de la clase Resultados

```
public class Resultados : ObservableCollection<Partido>
{
    public Resultados()
    {
        Add(new Partido("Osasuna",1,"Real Madrid", 4, "12/11/2006", "Liga"));
        Add(new Partido("Real Madrid", 5, "Écija", 1, "9/11/2006", "Copa"));
        Add(new Partido("Real Madrid", 1, "Celta", 2, "5/11/2006", "Liga"));
        Add(new Partido("Real Madrid", 1, "Steaua B.", 0, "1/11/2006",
"Champions"));
        Add(new Partido("Nástic", 1, "Real Madrid", 3, "28/10/2006", "Liga"));
        Add(new Partido("Écija", 1, "Real Madrid", 1, "25/10/2006", "Copa"));
        Add(new Partido("Real Madrid", 2, "Barcelona", 0, "22/10/2006",
"Champions"));
        Add(new Partido("Steaua B.", 1, "Real Madrid", 4, "17/10/2006",
"Champions"));
        Add(new Partido("Getafe", 1, "Real Madrid", 0, "14/10/2006", "Liga"));
        Add(new Partido("Real Madrid", 1, "Atlético", 1, "1/10/2006", "Liga"));
        Add(new Partido("Real Madrid", 5, "Dinamo K.", 1, "26/9/2006",
"Champions"));
        Add(new Partido("Betis", 0, "Real Madrid", 1, "23/9/2006", "Liga"));
        Add(new Partido("Real Madrid", 2, "R. Sociedad", 0, "17/9/2006", "Liga"));
        Add(new Partido("O. Lyon", 2, "Real Madrid", 0, "13/9/2006",
"Champions"));
        Add(new Partido("Levante", 1, "Real Madrid", 4, "10/9/2006", "Liga"));
        Add(new Partido("Real Madrid", 0, "Villareal", 0, "27/8/2006", "Liga"));
    }
}
```

```
 }//////////*//////
```

```
public class Partido
{
    private string equipoVisitador;
    private string equipoLocal;
    private int golesEquipoVisitador;
    private int golesEquipoLocal;
    private string fecha;
    private string competición;

    public Partido(string equipoVisitador, int golesEquipoVisitador,
                   string equipoLocal, int golesEquipoLocal,
                   string fecha, string competición)
    {
        this.equipoVisitador = equipoVisitador;
        this.equipoLocal = equipoLocal;
        this.golesEquipoVisitador = golesEquipoVisitador;
        this.golesEquipoLocal = golesEquipoLocal;
        this.fecha = fecha;
        this.competición = competición;
    }
    public string Ganador
    {
        get{
            if (golesEquipoLocal == golesEquipoVisitador)
                return "Empate";
            return golesEquipoVisitador > golesEquipoLocal ?
                equipoVisitador : equipoLocal; }
    }
    public string Fecha
    {
        get{ return fecha; }
    }
    public string Equipos
    {
        get { return equipoVisitador + " - " + equipoLocal; }
    }

    public string Resultado
    {
        get { return golesEquipoVisitador.ToString() + " - "
              + golesEquipoLocal.ToString(); }
    }
}
```

```

    }

    public string Competición
    {
        get { return competición; }
    }
}

```

Listado 13- 4 Código XAML, DataTrigger en un objeto .NET

```

<Window x:Class="DataTriggers.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:y="clr-namespace: DataTriggers"
    Title="Triggers: DataTrigger" Height="300" Width="300" >
<Window.Resources>
<y:Resultados x:Key="Resultados"/>
<Style TargetType="{x:Type ListViewItem}">
<Setter Property="Margin" Value="0,1,0,0"/>
<Setter Property="Height" Value="21"/>
<Style.Triggers>
<DataTrigger      Binding="{BindingPath=Ganador}"Value="Real
Madrid">
<Setter Property="Background">
<Setter.Value>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<LinearGradientBrush.GradientStops>
<GradientStop Color="#0E4791" Offset="0"/>
<GradientStop Color="#468DE2" Offset="1"/>
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Setter.Value>
</Setter>
<Setter Property="Foreground" Value="White" />
</DataTrigger>
</Style.Triggers>
</Style>
</Window.Resources>
<ListView ItemsSource="{Binding Source={StaticResource Resultados}}">
<ListView.View>
<GridView>

```

```

<GridViewColumn Header="Fecha" CellTemplate="{StaticResource Fecha}"/>
<GridViewColumn Header="Competición"
    CellTemplate="{StaticResource Competición}"/>
<GridViewColumn Header="Partido"
    CellTemplate="{StaticResource Partido}"/>
<GridViewColumn Header="Resultado"
    CellTemplate="{StaticResource Resultado}"/>
</GridView>
</ListView.View>
</ListView>
</Window>

```

Fíjese que el estilo que incluye el DataTrigger se ha asociado a los elementos de tipo ListViewItem. Un ListViewItem según el enlace a datos que se hace en este código, es de tipo Partido al que se aplica por tanto la propiedad Ganador que devuelve un string. Todo esto es lo que está detrás de la condición

```
<DataTrigger Binding="{Binding Path=Ganador}" Value="Real Madrid">
```

El efecto visual indicado en el DataTrigger es el de cambiar el color de fondo de cada ListViewItem que cumpla la condición de manera que todos los juegos ganados por el "Real Madrid" se visualicen como se muestra en la Figura 13-3.

Fecha	Competición	Partido	Resultado
12/11/2006	Liga	Osasuna - Real Madrid	1 - 4
9/11/2006	Copa	Real Madrid - Écija	5 - 1
5/11/2006	Liga	Real Madrid - Celta	1 - 2
1/11/2006	Champions	Real Madrid - Steaua B.	1 - 0
28/10/2006	Liga	Nástic - Real Madrid	1 - 3
25/10/2006	Copa	Écija - Real Madrid	1 - 1
22/10/2006	Champions	Real Madrid - Barcelona	2 - 0
17/10/2006	Champions	Steaua B. - Real Madrid	1 - 4
14/10/2006	Liga	Getafe - Real Madrid	1 - 0
1/10/2006	Liga	Real Madrid - Atlético	1 - 1
26/9/2006	Champions	Real Madrid - Dinamo K.	5 - 1
23/9/2006	Liga	Betis - Real Madrid	0 - 1
17/9/2006	Liga	Real Madrid - R. Sociedad	2 - 0
13/9/2006	Champions	O. Lyon - Real Madrid	2 - 0
10/9/2006	Liga	Levante - Real Madrid	1 - 4
27/8/2006	Liga	Real Madrid - Villarreal	0 - 0

Figura 13 - 3 Estadística de los partidos ganados por el Real Madrid

Al igual que con los Trigger cuando deja de cumplirse la condición las propiedades modificadas por el DataTrigger vuelven a tomar su valor anterior.

Con en el ejemplo anterior se ha ilustrado cómo con el DataTrigger se puede monitorear el valor de cualquier propiedad de cualquier objeto de nuestra aplicación y producir efectos en la apariencia según esta propiedad cambie. Por supuesto que con un DataTrigger también podemos monitorear una propiedad de un control del propio WPF (a fin de cuentas un control

WPF es un objeto .NET). El Listado 13-5 nos muestra el uso de DataTrigger para cambiar la propiedad Source de un Image a partir de la selección de un determinado control CheckBox (Figura 13-4).

Listado 13- 5 Código XAML, DataTrigger en un control WPF

```
<Window x:Class="DataTriggers.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Triggers: DataTrigger" Height="300" Width="300"
    Background="DarkBlue" >
<Window.Resources>
    <Style TargetType="{x:Type Image}">
        <Setter Property="Image.Source" Value="WinditoAlegre.gif"/>
        <Style.Triggers>
            <DataTriggerBinding="{BindingElementName=RB1, Path=IsChecked}" Value="True">
                <Setter Property="Image.Source" Value="WinditoAlegre.gif"/>
            </DataTrigger>
            <DataTriggerBinding="{BindingElementName=RB2, Path=IsChecked}" Value="True">
                <Setter Property="Image.Source" Value="WinditoDudoso.gif"/>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=RB3, Path=IsChecked}" Value="True">
                <Setter Property="Image.Source" Value="WinditoTriste.gif"/>
            </DataTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
<StackPanel>
    <Image Name="Imagen" Height="200">
        </Image>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <RadioButton IsChecked="True" Name="RB1" FontFamily="Wingdings" FontSize="50"
FontWeight="Bold" Foreground="White">
                J
            </RadioButton>
            <RadioButton Name="RB2" FontFamily="Wingdings" FontSize="50"
FontWeight="Bold" Foreground="White">
                K
            </RadioButton>
            <RadioButton Name="RB3" FontFamily="Wingdings" FontSize="50"
FontWeight="Bold" Foreground="White">
                L
            </RadioButton>
        </StackPanel>
    </Image>
</StackPanel>
```

```
</StackPanel>  
</StackPanel>  
</Window>
```



Figura 13 - 4 Cambiando el Source del Image con un DataTrigger

Al igual que con los Trigger se pueden agrupar más de una condición para un DataTrigger usando un MultiDataTrigger

13.3 EventTriggers

Mientras los Trigger preguntan por los valores de propiedades de los propios elementos y los DataTrigger por valores de propiedades de cualquier objeto .NET (al que se puede referir por los enlaces a datos), hay una tercera categoría de desencadenadores que son los EventTrigger que chequean la ocurrencia de eventos.

Los EventTrigger están muy relacionados con las capacidades de animaciones de WPF.

Todos los FrameworkElement tienen una propiedad Triggers (una colección de objetos TriggerBase) la cual tiene un sentido especial. A pesar de ser una colección de elementos TriggerBase solo pueden contener objetos de tipo EventTrigger.

Los EventTrigger definen tres propiedades fundamentales:

SourceName de tipo string que se refiere al nombre del elemento (que se le asocia al elemento con la propiedad Name o x:Name) sobre el que se trabaja. En la práctica la mayoría de las veces no es necesario especificar valor a esta propiedad, es definida por el propio contexto del EventTrigger.

La propiedad RoutedEvent contiene el nombre del evento que provocará el “desencadenado” de las acciones definidas en el EventTrigger.

Actions es la propiedad que define el conjunto de acciones a desencadenar cuando ocurra el evento especificado como condición del EventTrigger. La propiedad Actions es una colección de objetos TriggerAction. La clase derivada de TriggerAction en WPF más importante es quizás BeginStoryboard que será vista con más detalle en la Lección **Animaciones**.

Un EventTrigger desencadenará la ejecución de un conjunto de acciones cuando ocurra un determinado Evento Enrutado (RoutedEvent). Por ejemplo podemos usar un EventTrigger para comenzar una animación cuando el cursor del ratón se encuentre sobre algún control de la interfaz de usuario. No confunda esto con el caso de un desencadenador Trigger definido de la forma <Trigger Property="IsMouseOver" Value="True">, en este último caso el cambio desencadenado por el Trigger se deshace en cuanto la propiedad IsMouseOver deje de ser True. Aunque se parece esto no es lo mismo que si asociamos un EventTrigger al evento MouseOver A diferencia de los Trigger y DataTrigger, los EventTrigger no tienen concepto de terminación. El conjunto de acciones se desencadena cuando ocurre el evento y las propiedades que se modifiquen no retoman implícitamente el valor anterior porque no hay concepto de que el **evento se acabó**. Si Ud. quiere lograr un efecto de que algunos valores de la apariencia vuelvan a algún estado anterior debe cuidar de programar explícitamente esto de algún modo.

Veamos el ejemplo de una ventana en la que hemos colocado una imagen a la cual se le modificará el valor de su propiedad Height en el EventTrigger en respuesta a la ocurrencia de un determinado evento. Note que para el evento MouseEnter, aumentamos el valor de la propiedad Height logrando un efecto de "zoom" como ilustra la Figura 13-5.

El proceso inverso al desencadenado hay que programarlo explícitamente, recuerde que con los EventTriggers las propiedades no retoman sus valores originales de forma automática, es por eso que Ud. notará en el Listado 13-6 otro EventTrigger que responde al evento MouseLeave y que es el encargado de retornar las propiedades modificadas (en este caso la propiedad Height de la imagen) a su valor original.

Listado 13- 6 Código XAML, EventTrigger

```
<Window x:Class="EventTriggers.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Triggers: EventTrigger" Height="300" Width="300" Name="Ventana"
    Background="DarkBlue" >
<Window.Resources>
<Style TargetType="{x:Type Image}">
    <Style.Triggers>
        <EventTrigger RoutedEvent="Image.MouseEnter">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimationTo="300" Duration="0:0:1.5" AccelerationRatio="0.10"
DecelerationRatio="0.25" Storyboard.TargetProperty="(Image.Height)" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Style.Triggers>
</Style>

```

```

</EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Image.MouseLeave">
<EventTrigger.Actions>
<BeginStoryboard>
<Storyboard>
<DoubleAnimation Duration="0:0:1.5"
AccelerationRatio="0.10" DecelerationRatio="0.25"
Storyboard.TargetProperty="(Image.Height)" />
</Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Style.Triggers>
</Style>
</Window.Resources>
<Image Height="100" HorizontalAlignment="Center" VerticalAlignment="Center"
Source="Windito.gif"/>
</Window>

```

La Figura 13-5 ilustra una secuencia del efecto “zoom” que provoca la animación desencadenada por el EventTrigger cuando el cursor del ratón entra en el área de la imagen. Note que al salir del área de la imagen se desencadenará entonces una animación en sentido inverso (porque la programamos explícitamente) hasta que el Height retome su valor.



Figura 13 - 5 Cambio del valor de la propiedad Height de la imagen por un EventTrigger

Como Ud. habrá podido apreciar cuando de cambios visuales y de apariencia se trata el uso de desencadenadores nos permite prescindir del patrón de evento y manejador de evento programado en el code-behind, lo cual debe quedar solo cuando esto implique la ejecución de acciones de la lógica del negocio no expresable en la naturaleza declarativa de XAML.

Lección 14 Plantillas de Controles

En las lecciones del capítulo sobre gráficos aprendimos cómo dibujar usando figuras, brochas y pinceles; y vimos también cómo aplicar transformaciones y efectos visuales a los elementos de interacción. En la Lección **Plantillas de Datos** vemos cómo definir plantillas para expresar apariencias con las que mostrar datos. En esta lección veremos cómo usar todos estos recursos de dibujo para modificar la apariencia de los controles que usted usa en su aplicación. También en esta lección aplicaremos los contenidos aprendidos en las lecciones de enlace a datos y triggers.

En las plataformas de controles anteriores a WPF como MFC, Windows.Form o Web.Form, la apariencia de los controles estaba estrechamente vinculada al propio control. Cambiando el valor de algunas de sus propiedades como BackColor, ForeColor o Font se podían lograr ligeras modificaciones en el aspecto de los controles. En los controles de WPF también se puede hacer algo parecido modificando propiedades como Background, Foreground, FontSize, FontWeight, etc. Sin embargo quisiéramos más porque estos cambios no cubren todos los aspectos de apariencia. ¿Qué tal si en vez de querer que un botón tenga un color de fondo diferente, queremos que fuese redondo? Para lograr esto en Windows.Form usted tendría que implementar su propio tipo de botón y redefinir el método OnPaint de la clase Control para dibujar la forma redondeada del botón cuando éste fuese repintado. Pero incluso aún cuando lo dibuje redondo el área activa del botón seguiría siendo rectangular y para cambiar esto usted tendría además que calcular una región diferente para la ventana Win32 que soportaba al control. Todos estos dolores de cabeza se eliminan o disminuyen con la nueva estructura que tienen los controles de WPF.

Una de las más atractivas novedades de WPF es el uso de plantillas (templates) que se pueden escribir en XAML y que nos permiten encapsular la apariencia de un control. Todos los controles en WPF son definidos en dos partes: Su funcionalidad, que se expresa por ser basado en un tipo heredero de Control (y que se escribe en cualquier lenguaje .NET como C#) y su apariencia, que se expresa en una plantilla del control que es un tipo derivado de ControlTemplate (y que suele definirse enteramente en XAML). De este modo la forma en que se muestra visualmente un control está enteramente descrita por su plantilla visual la cual puede ser cambiada por otra plantilla cuando se deseé. Gracias a esto se podría entonces lograr un botón redondo modificando sólo la plantilla visual del mismo sin tener que tocar el código que define su funcionalidad. A través de la plantilla se podría indicar entonces que la figura que describe la apariencia del control fuese una elipse, o un rectángulo de esquinas redondeadas. Como caso de ejemplo en esta lección describiremos paso a paso la definición de una plantilla para que los botones puedan lucir redondos y cristalinos.

14.1 Plantillas visuales

En Windows.Forms Ud. puede cambiar el color de un control del mismo modo que podría cambiar el color de su coche: pintándolo de otro color. En WPF usted puede lograr el efecto de cambiarle a su coche no solo el color sino la carrocería completa. Una plantilla visual para un control WPF viene a ser como la carrocería para un coche (Figura 14-1). Sin ella el coche es en principio funcional: arranca, gira, acelera, corre, frena, etc. pero no es atractivo. En WPF usted podrá cambiarle la apariencia a los controles que desee sin necesidad de definir nuevos controles (algo así como cambiar la carrocería sin tener que comprarse otro coche). Hay que decir que lamentablemente que en la irracionalidad actual muchos cambian de coche solo por cambiar de carrocería aún cuando el motor y el resto del mecanismo sigan funcionando de maravilla.



Figura 14-1 Control Coche

Los controles de WPF se componen de dos elementos principales: su lógica funcional y su plantilla visual. La parte funcional se ocupa de mantener el estado del control y la interactividad con la lógica de negocio que lo esté utilizando, mientras que la plantilla visual se ocupa de darle color y otros efectos que le den atractivo y facilidad de uso. Con esta nueva estructura en el esquema de definición de controles de WPF usted puede cambiar fácilmente la apariencia de cualquier control con sólo cambiar su plantilla.

Todo control WPF tiene por defecto una plantilla que lo representa visualmente. Un control sin plantilla simplemente no se ve.

Las plantillas en WPF son objetos de tipo `ControlTemplate`. Una plantilla está "ligada" con un control a través de la propiedad `Template` que tienen todos los objetos derivados de la clase `Control`. Usted puede modificar la propiedad `Template` de un control asignándole una plantilla que se define directamente como se ilustra en el Listado 14-1, sin embargo esto no es lo más aconsejable. Usualmente queremos reutilizar una plantilla para que varios controles compartan un mismo estilo visual, es decir que tengan un aspecto parecido. Recordemos que en la Lección **Estilos** comentamos que la manera por excelencia de compartir estilos visuales es mediante el empleo de estilos que se colocan en los recursos de los elementos que componen la aplicación. El Listado 14-2 nos muestra un esquema de definición de plantillas que mantendremos en todos los ejemplos de plantillas de esta lección.

Listado 14-1 Definición directa de una plantilla para un botón

```

<Button>
  <Button.Template>
    <ControlTemplate>
      ...
      </ControlTemplate>
    </Button.Template>
  </Button>

```

Listado 14-2 Plantilla definida en un estilo de botones

```

<Page ...>
  <Page.Resources>
    <Style TargetType="{x:Type Button}" x:Key="ButtonGlass">
      <Setter TargetType="Template">
        <Setter.Value>
          <ControlTemplate ...>
            ...
            </ControlTemplate>
          </Setter.Value>
        </Setter>
      </Style>
    </Page.Resources>
    <Button Style="{StaticResource ButtonGlass}" />
  </Page>

```

El Listado 14-2 nos muestra cómo definimos un estilo `ButtonGlass` (botón cristalino) para aplicar a botones en el cual se indica cambiarle el valor a la propiedad `Template` de un botón por el objeto `ControlTemplate` que se define inline en el propio XAML (y que por brevedad ahora en el código se ha denotado con ...). Vamos a destacar aquí dos aspectos primordiales:

1. Una propiedad `TargetType` a través de la cual debe indicarse a qué tipo de control se aplica la plantilla. Esto permite luego a WPF evitar que usted cometa el error de intentar ponerle una plantilla a un control que no tiene que ver con la plantilla (como querer ponerle una carrocería de ómnibus a un coche). Sin embargo, por medio de la herencia usted puede lograr flexibilidad porque si indica que el `TargetType` es un tipo base de una jerarquía (por ejemplo `ButtonBase`) luego usted podrá aplicar la misma plantilla tanto para `Button`, `ToggleButton` o cualquier otro tipo heredero de `ButtonBase`.

2. El contenido o valor ControlTemplate que le da a la propiedad Template es un elemento visual que define la apariencia que tendrá el control al que se le aplica la plantilla. Dicho de otra manera, este elemento visual será exactamente lo que usted verá en su monitor cuando WPF despliegue el control. En WPF cuando usted vea un botón u otro control, no se engañe. Usted realmente está identificando a un borde redondeado, coloreado con gradientes y un texto como apariencia del botón, pero si usted pusiera un botón sin plantilla entonces no vería absolutamente nada, ni obtendría tampoco ningún evento de teclado o ratón. El botón no es un elemento gráfico, es sólo el motor. Un control por sí mismo no se ve, lo que usted ve y con lo que usted interactúa es con los elementos que componen su plantilla: la carrocería. Usted hace clic sobre la carrocería y es el motor el que se encarga de provocar la reacción ante este evento.

El Listado 14-3 nos muestra una plantilla para lograr un estilo de botón cristalino con esquinas redondeadas en el que se indica que de tener este un contenido este se muestre en el centro del botón. Este estilo con plantilla se le ha asociado luego a un botón de contenido Cerrar que se incluye dentro de un Canvas. Para mayor visibilidad del efecto cristalino del botón hemos puesto el fondo de la página con un rayado (como puede apreciarse en la Figura 14-2). Por brevedad no se incluye esta parte del código.

Listado 14-3 Plantilla de botón definida con Borders

```
<Page x:Class="ControlTemplates.GlassButtonTemplate"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="GlassButtonTemplate">
<Page.Resources>
  <Style TargetType="{x:Type Button}" x:Key="GlassButton">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
          <Border CornerRadius="10"
                  BorderBrush="{TemplateBinding BorderBrush}"
                  BorderThickness="{TemplateBinding BorderThickness}">
            <Grid>
              <BorderBackground="{TemplateBinding Background}">
                <CornerRadius="10">
                  <Border.OpacityMask>
                    <LinearGradientBrush EndPoint="0,1">
                      <GradientStop Color="#aFFF" Offset="0"/>
                      <GradientStop Color="#2FFF" Offset="1"/>
                    </LinearGradientBrush>
                </CornerRadius>
              </BorderBackground>
            </Grid>
          </Border>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Page.Resources>
<Canvas>
  <ContentControl Content="Cerrar" Style="{StaticResource GlassButton}"/>
</Canvas>

```

```

        </Border.OpacityMask>
    </Border>
    <Border Margin="4,1" CornerRadius="5,5,0,0" Height="5"
        VerticalAlignment="Top">
        <Border.Background>
            <LinearGradientBrush EndPoint="0,1">
                <GradientStop Color="#dfff" Offset="0"/>
                <GradientStop Color="Transparent" Offset="1"/>
            </LinearGradientBrush>
        </Border.Background>
    </Border>
    <ContentPresenter VerticalAlignment="Center"
        HorizontalAlignment="Center"/>
</Grid>
</Border>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Page.Resources>
<Page.Background>
    ...
</Page.Background>
<Canvas>
    <Button Canvas.Left="10" Canvas.Top="10" Width="90" Height="40"
        Style="{StaticResource GlassButton}" FontSize="16"
        BorderThickness="1.3" Background="RoyalBlue">
        Cerrar
    </Button>
</Canvas>
</Page>

```



Figura 14-2 Botón con estilo GlassButton

Como se ve en la Lección **Figuras** el Border es uno de los elementos más usados en la definición de los estilos visuales de los controles en WPF. La plantilla que define el estilo del botón del Listado 14-3 está hecha a base de elementos Border y de un Grid. Observe que la redondez de las esquinas del botón se obtiene a través de la propiedad CornerRadius del Border que hemos

puesto como raíz del contenido de la plantilla y que el estilo cristalino se consigue con la propiedad OpacityMask del primer Border interno del control. Además hemos empleado un segundo Border interno puesto convenientemente en la parte superior del Grid con un gradiente lineal semitransparente para obtener el efecto de brillo en el botón. En el Listado 14-3 se ha hecho uso de dos nuevos tipos que son de gran utilidad en las plantillas de controles: TemplateBinding y ContentPresenter y que veremos en las secciones a continuación.

14.1.1 TemplateBinding

En el Listado 14-3 note el valor que hemos asignado a las propiedades BorderBrush y BorderThickness del Border más externo y en el primer Border interno el valor de la propiedad Background. En estos tres casos notará que hemos empleado la extensión de marca TemplateBinding. Esta extensión es un tipo particular de Binding que se emplea para enlazar las propiedades de los elementos que componen la plantilla con propiedades del objeto al que se le aplique esta plantilla. De no hacer enlaces como estos usted estaría utilizando los valores predeterminados de estas propiedades definidas en el propio Control o en su la plantilla por defecto. Más abajo en el mismo Listado 14-3 hemos puesto un botón con el estilo GlassButton que es el que usted ve en la Figura 14-2. Observe que en la definición del botón le hemos dado valor "1.3" a su propiedad BorderThickness. Como esta propiedad ha quedado enlazada con la propiedad BorderThickness del Border, al asignar un valor al grosor de borde del botón el grosor del Border también tendrá este mismo valor. Fíjese que lo mismo ocurre con la propiedad Background a la que hemos dado valor RoyalBlue. En el Listado 14-4 hemos puesto dos botones con el mismo estilo y con diferentes grosores para sus bordes. Note en la Figura 14-3 como cada valor produce un grosor diferente en el mismo estilo para los dos botones.

Listado 14-4 Dos botones con grosor de borde diferente
<pre><Page ...> <Page.Resources> <Style TargetType="{x:Type Button}" x:Key="GlassButton"> <Setter Property="Template"> <Setter.Value> <ControlTemplate TargetType="{x:Type Button}"> <Border ... BorderThickness="{TemplateBinding BorderThickness}"> ... </Border> </ControlTemplate> </Setter.Value> </Setter> </Style> </Page.Resources> <Canvas></pre>

```

<Button ... BorderThickness="1.3" Style="{StaticResource GlassButton}">
    Cerrar
</Button>
<Button ... BorderThickness="3"     Style="{StaticResource GlassButton}">
    Cerrar
</Button>
</Canvas>
</Page>

```



Figura 14-3 Botones con el mismo estilo y diferente grosor de borde

14.1.2 ContentPresenter

Al describir la plantilla usted normalmente pone figuras y demás elementos visuales que den el efecto visual que se desea para el aspecto del control. Sin embargo, cuando usted quiera que la plantilla se aplique a diferentes controles de un mismo tipo, es necesario mostrar algunos datos particulares de cada uno. Si bien TemplateBinding nos permite hacer enlaces con propiedades del control al que se aplique la plantilla, ContentPresenter es una manera particular de enlazar con el contenido del control cuando este sea de un tipo heredero de ContentControl. El botón es un ContentControl y como tal tiene un contenido que se puede indicar explícitamente en su interior (poniéndolo entre las marcas `<Button>` y `</Button>`) o asignándolo explícitamente a la propiedad `Content` que tienen todos los herederos de ContentControl. Al poner un ContentPresenter dentro de la plantilla este se encargará de ubicar el contenido en la posición que se indique dentro de la plantilla. Si en una plantilla como la del Listado 14-3 se omite poner el ContentPresenter se obtendría como resultado un botón cristalino y de esquinas redondeadas pero sin contenido.

14.1.3 Triggers

Recordemos que las plantillas contienen lo que realmente se muestra en la pantalla cuando se muestre el control. Lo que se desplegará en el área visual de la ventana son sólo las figuras y demás elementos visuales que se hayan puesto en su plantilla. ¿Cómo hace entonces el botón para disparar el evento Click cuando se presione el ratón con el cursor encima de él? Cuando hacemos clic sobre el área del botón lo que realmente se está provocando son eventos de ratón sobre las figuras que componen su plantilla. El botón "se entera" de la acción de presión del

ratón gracias al mecanismo de *Bubbling* y *Tuneling* (vea Lección **Eventos y Comandos**) de los eventos de WPF y hace que se dispare un evento Click.

Las diferentes acciones que se aplican sobre un botón generan cambios en su apariencia. Experimente con los botones de Windows y observe como estos botones varían ligeramente su aspecto cuando usted pasa el ratón por encima de ellos o cuando retiene el ratón apretado sobre alguno. La mayoría de estos eventos están reflejados en los controles a través de diferentes propiedades que nos indican cuándo el control debe cambiar su estado visual. En el caso del botón usted puede saber cuándo el ratón está encima a través de la propiedad IsMouseOver, cuándo el botón está presionado mediante IsPressed o cuando el control está o no habilitado consultando a IsEnabled. En la plantilla de un control usted puede interrogar el valor de estas propiedades usando desencadenadores (triggers) para definir diferentes apariencias para cada uno de estos estados.

En el Listado 14-5 tenemos en la plantilla del botón un Trigger que se aplica cuando el control esté deshabilitado. Note que en la propiedad Property del Trigger se ha indicado que la propiedad que lo desencadena es IsEnabled cuando esta tome el valor False, que se indica como valor de su propiedad Value. Como contenido de este elemento Trigger hay tres Setters que se aplican cuando el trigger se desencadene. Los primeros dos Setters hacen que el Foreground del botón tome valor DimGray y el BorderBrush tome valor DarkGray. Como usted puede observar en la Figura 14-4 estos dos cambios se ven reflejados en el color del contorno del borde más externo del botón gracias a TemplateBinding y en el texto gracias ContentPresenter.



Figura 14-4 Botones con estilo GlassButton. El segundo está deshabilitado

Como el estado visual de un control se expresa mediante los elementos que se ponen en la plantilla, algunos de los Setters pueden entonces tomar como objetivo de sus acciones a uno o a varios de estos elementos. El tercer Setter por ejemplo no se aplica sobre una propiedad del botón sino sobre la propiedad Background del Border de nombre BackBorder. Cuando usted quiera que algún Setter se aplique a una propiedad de alguno de los elementos de la propia plantilla, entonces debe haberle dado nombre al elemento (lo que se hizo con Name="BackBorder"), de este modo luego puede indicarle al Setter que este es su elemento objetivo mediante la propiedad TargetName como se muestra Listado 14-5.

Listado 14-5 Trigger aplicado a la plantilla de un botón para su estado: Deshabilitado

<Page ...>
<Page.Resources>

```

<Style TargetType="{x:Type Button}" x:Key="GlassButton">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                <Border ...>
                    <Grid>
                        <Border Background="{TemplateBinding Background}" CornerRadius="10"
Name="BackBorder">
                            <Border.OpacityMask>
                                <LinearGradientBrush EndPoint="0,1">
                                    <GradientStop Color="#aFFF" Offset="0"/>
                                    <GradientStop Color="#2FFF" Offset="1"/>
                                </LinearGradientBrush>
                            </Border.OpacityMask>
                        </Border>
                        ...
                    </Grid>
                </Border>
                <ControlTemplate.Triggers>
                    <Trigger Property="IsEnabled" Value="False">
                        <Setter Property="Foreground" Value="DimGray"/>
                        <Setter Property="BorderBrush" Value="DarkGray"/>
                        <Setter TargetName="BackBorder" Property="Background" Value="Gray"/>
                    </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
</Page.Resources>
<Canvas>
    <Button ... BorderThickness="1.3">
        Cerrar
    </Button>
    <Button ... BorderThickness="1.3" IsEnabled="False">
        Cerrar
    </Button>
</Canvas>
</Page>

```

Entre las marcas <ControlTemplate.Triggers> y </ControlTemplate.Triggers> usted puede poner tantos Triggers como crea conveniente para expresar los diferentes estados visuales de un control pero tenga en cuenta que los Triggers se van desencadenando en el orden que aparecen. Por ejemplo si usted define dos Triggers uno para el estado "ratón encima" (IsMouseOver) y otro para "Presionado" (IsPressed) los dos se desencadenarán cuando usted presione sobre el botón porque se cumple a la vez que el ratón estará sobre el botón mientras lo presiona, pero si el que representa al estado "Presionado" aparece de último en la lista (como se muestra en el Listado 14-6) entonces los efectos de este último son los que se visualizarán de último lo que puede significar que se dejen de apreciar efectos anteriores.



Listado 14-6 Nuevos triggers que se aplican cuando el botón tiene el ratón encima o está presionado

```
<Page ...>
<Page.Resources>
<Style TargetType="{x:Type Button}" x:Key="GlassButton">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Button}">
<Border ... Name="OuterBorder">
<Grid>
<Border ... Name="BackBorder">
...
</Border>
...
</Border>
<ControlTemplate.Triggers>
<Trigger Property="IsEnabled" Value="False">
...
</Trigger>
<Trigger Property="IsMouseOver" Value="True">
<Setter TargetName="OuterBorder" Property="Background"
Value="#9FFF"/>
</Trigger>
<Trigger Property="IsPressed" Value="True">
<Setter TargetName="OuterBorder" Property="Background"
Value="#4000"/>
<Setter TargetName="BackBorder" Property="OpacityMask">
<Setter.Value>
<LinearGradientBrush EndPoint="0,1">
<GradientStop Color="#dFFF" Offset="0"/>
<GradientStop Color="#6FFF" Offset="1"/>
```

```

</LinearGradientBrush>
</Setter.Value>
</Setter>
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Page.Resources>
<Canvas>
<Button ...>
    Cerrar
</Button>
<Button ...>
    Cerrar
</Button>
</Canvas>
</Page>

```

En la plantilla del Listado 14-6 se le ha dado nombre también al Border más externo de la plantilla cuando se hace Name="OuterBorder" luego este nombre se usa por el Trigger para indicar que la propiedad Background se modifica cuando el botón tenga el ratón encima o cuando esté presionado. Observe que los dos nuevos Triggers tienen Setters que afectan a la misma propiedad Background del OuterBorder. El primero asigna una brocha sólida de color blanco semitransparente que da el efecto de perder un poco la transparencia del efecto cristalino que tenía el botón, el segundo Setter hace que cuando el botón esté presionado se oscurezca el fondo al aplicar una brocha negra semitransparente. Recuerde que cuando el botón deje de estar presionado (se suelte la presión sobre el ratón) la condición de este desencadenador se dejará de cumplir y según lo explicado en la Lección **Triggers** las propiedades modificadas volverán a su valor anterior. En la Figura 14-5 se muestran estos cuatro estados visuales para el botón. Usted podrá percibir que en el estado presionado el fondo del control es más oscuro que en los demás estados, lo que indica que el último Trigger fue el que prevaleció sobre la propiedad Background del elemento OuterBorder cuando se presionó el botón.



Figura 14-5 Los cuatro estados del estilo GlassButton

En lugar del elemento Trigger se puede usar EventTrigger si desea iniciar alguna animación cuando ocurra un evento. En la Lección **Animaciones** usted podrá encontrar ejemplos que ilustran cómo hacerlo.

14.2 Plantillas de controles compuestos

Desde que se comenzaron a usar interfaces de usuario gráficas y ventanas existen los llamados controles compuestos. Un ejemplo de ellos es el Scrollbar, éste es un control que contiene dos botones en los extremos y un tarequito (*thumb*) que se arrastra para desplazar. El Slider es otro control parecido que se muestra normalmente como una barra con un *thumb* que permite seleccionar un valor numérico en un rango (Figura 14-6).



Figura 14-6 Estilo por defecto del Slider

En esta sección mostraremos como cambiar el estilo del Slider para que se vea con una apariencia como la que se utiliza en un Media Player (Figura 14-7). Esta es la apariencia que se utilizará en la aplicación **Visor de Curso** que sirve de ejemplo general de este curso.



Figura 14-7 Slider que actúa como control de volumen en la aplicación de ejemplo del curso

Repasemos primero cómo se interactúa con este control. Existen tres formas de cambiar el valor del Slider usando el ratón:

1. Arrastrando el *thumb* hasta la posición que nos acomode.
2. Haciendo clic a la izquierda del *thumb* para hacerlo disminuir el valor paso a paso.
3. Haciendo clic a la derecha para incrementar el valor paso a paso.

También podemos mantener el ratón presionado en cualquiera de ambos lados del thumb haciendo que el *thumb* se mueva continuamente paso a paso hasta la posición que querramos.

Si en vez de tener un Slider quisieramos lograr toda esta funcionalidad con controles independientes podemos simular el Slider con un elemento tipo Thumb y dos RepeatButtons a cada lado del Thumb (Figura 14-8). La funcionalidad del Slider pudiera simularse haciendo que en cada clic de los RepeatButtons se moviera el Thumb a la izquierda o la derecha en dependencia de cual RepeatButton presionemos, así conseguiríamos la funcionalidad esperada

para las formas de interacción 2 y 3 descritas anteriormente. Si además hicéramos que el Thumb se moviera sólo horizontalmente en un rango y pusiéramos a los tres controles en un StackPanel haciendo que cada RepeatButton tenga HorizontalAlignment = "Stretch" para que cambiaren de tamaño al moverse el Thumb estaríamos obteniendo la primera de las formas de interacción anteriores.

Pues la buena noticia es que toda esta funcionalidad la tiene ya implementada el Slider, junto con el cambio de su valor en cada acción, justamente a partir de dos RepeatButtons y un Thumb. Por lo que cambiar la apariencia del Slider se traduce entonces en cambiar la apariencia de dos RepeatButtons y un Thumb. Pero todavía tenemos que entender algo más antes de pasar a definir su plantilla...



Figura 14-8 Dos RepeatButtons a ambos lados de un Thumb que funcionan como un Slider.

Tómese un tiempo ahora para estudiar el comportamiento del Scrollbar... Advertirá que excluyendo los botones de los extremos un Scrollbar funciona muy parecido a un Slider ya que tiene un Thumb para desplazar y dos RepeatButtons que realizan las mismas operaciones que en el Slider. Ambos aprovechan que tienen una funcionalidad común tan similar y utilizan un tercer "elemento más primitivo" común para ambos. Este elemento es un Track donde se ha encapsulado toda la lógica de interacción que recién hemos descrito.

14.2.1 El track

El Track no es realmente un control, no hereda de Control ni tiene una plantilla visual asociada. Pero sí es un elemento de interacción que hereda de FrameworkElement por lo que puede recibir eventos de ratón y teclado aunque no se presente visualmente de modo alguno.

El Track no sólo es un elemento creado para encapsular una lógica común entre diferentes controles sino que además es un ejemplo vivo de elementos que podemos ubicar en la frontera entre la funcionalidad de un control y su presentación visual.

¿Sabía usted que el TextBox, uno de los controles más comunes, contiene también elementos frontera? Intente cambiar su plantilla y notará que también en el caso del TextBox usted necesitará utilizar un elemento frontera que se encargue de las operaciones de edición del texto.

Si se anima a experimentar con cambiar estilos encontrará muchos más controles compuestos y con elementos frontera de los que usted imagina.

La funcionalidad de muchos controles consiste solamente en mantener propiedades que indiquen el estado visual de los controles y en poder realizar operaciones de cara a la lógica de

negocio que los utilice, sin embargo, hay controles compuestos como el Slider o el Scrollbar que incluyen a su vez funcionalidad vinculada con la forma en que se presentan los controles que los componen. ¿Cómo hacer entonces para mantener la metáfora de: carrocería por un lado y motor y mecanismo de transmisión por otro? La respuesta está en tener elementos que habiten en la frontera. El Slider por ejemplo "sabe" como interactuar con un Track que tenga en su plantilla y ajusta los valores del Track acorde con la posición del Thumb. Por su lado el Track permite que usted, desde una plantilla, modifique su Thumb y sus RepeatButtons poniendo el estilo visual que le convenga para cada uno de estos controles. Note que a diferencia de un botón, un Slider puede "conocer" que hay determinado tipo de elemento (un Track) en su plantilla con el cual él puede interactuar. Veremos a continuación cómo hacer para indicarle a un Slider cuál es el Track de su plantilla que debe tomar para su interacción.

14.2.2 Elementos frontera: PARTS

Anímese ahora a abrir la documentación y observe la definición de la clase Slider (Figura 14-9). Fíjese en el último atributo TemplatePartAttribute. Este atributo es el que le indica al Slider cuál es el nombre (PART_Track en este caso) que debe llevar el Track que se ponga en su plantilla. Note que además el atributo requiere que el elemento que lleve el nombre PART_Track debe ser de tipo Track (Type=typeof(Track)).

Nos referimos aquí a un atributo .NET que es un recurso de metainformación para asociar a elementos de código .NET. Lamentablemente en la terminología XAML también se utiliza el término atributo para hablar de las propiedades de un elemento, esto no debe confundir al lector porque ambos usos del mismo término no tienen relación entre sí.

En general la presencia de un atributo TemplatePartAttribute asociado a la definición de un tipo de control, nos permite conocer si el control contiene tales "elementos frontera" y nos dice qué nombre debemos darle a estos elementos cuando los pongamos en la plantilla para que el control pueda referirse a ellos. En la Figura 14-10 mostramos la documentación de la clase TextBoxBase que nos indica qué tipo de elemento (FrameworkElement) frontera debe ponerse en su plantilla para que controles como el TextBox lo empleen como anfitrión para la edición del texto.

Plantilla de un Slider sin Track

Aunque los elementos frontera son útiles para hacer que su control funcione de la manera habitual, usted no está obligado a ponerlos. Es posible definir una plantilla de TextBox o de Slider sin poner estos elementos y entonces queda completamente de parte suya el diseñar la plantilla con elementos que aprovechen todas las capacidades del control. El Listado 14-7 y la Figura 14-11 muestran el resultado de construir la plantilla de un Slider sin Track.

Listado 14-7 Plantilla de Slider sin Track

```

<Page x:Class="ControlTemplates.ThinSliderTemplate"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="ThinSliderTemplate" >
<Canvas>
<Canvas.Resources>
<Style TargetType="{x:Type Slider}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Slider}">
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="20"/>
<ColumnDefinition/>
<ColumnDefinition Width="20"/>
</Grid.ColumnDefinitions>
<RepeatButton Command="Slider.DecreaseLarge">
&lt;&lt;/RepeatButton>
<TextBox Grid.Column="1"
Text="{Binding RelativeSource={RelativeSource
TemplatedParent}, Path=Value,
Mode=TwoWay}"/>
<RepeatButton Command="Slider.IncreaseLarge"
Grid.Column="2">&gt;</RepeatButton>
</Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Canvas.Resources>
<Slider Canvas.Left="10" Canvas.Top="10" Width="120"
Height="25"/>
</Canvas>
</Page>

```

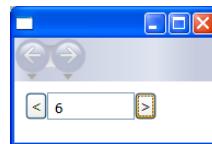


Figura 14-11 Slider con una plantilla sin Track

En la Figura 14-11 se puede observar un número (6) que se corresponde con el valor del Slider. Note en el Listado 14-7 que en vez de usar TemplateBinding para enlazar la propiedad Text del TextBox con la propiedad Value del control Slider al que se aplique la plantilla, hemos empleado directamente un enlace Binding. Este enlace es equivalente al TemplateBinding sólo que este enlaza a las propiedades en los dos sentidos (Mode=TwoWay) mientras que con TemplateBinding sólo podríamos actualizar el texto del TextBox con el valor del Slider y no en el sentido inverso (OneWay).

Plantilla del Slider con Track

Hechas todas estas observaciones ya estamos listos para confeccionar una plantilla que muestre un slider como el de la Figura 14-7. Para ello iremos construyendo la plantilla por partes.

El primer paso es definir la plantilla mostrando una barra que revele el espacio que cubre todo el control (Listado 14-8 y Figura 14-12).

Listado 14-8 Plantilla de Slider que sólo incluye una borde que revela su barra de rango

```
<Page ...>
<Canvas Background="LightSteelBlue">
<Canvas.Resources>
<Style TargetType="{x:Type Slider}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Slider}">
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"
MinHeight="{TemplateBinding MinHeight}"/>
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
<Border Height="5" Grid.Row="1" CornerRadius="2"
Background="Transparent" BorderThickness="1">
<Border.BorderBrush>
<LinearGradientBrush EndPoint="0,1">
<GradientStop Color="#4000" Offset="0"/>
<GradientStop Color="#dfff" Offset="1"/>
</LinearGradientBrush>
</Border.BorderBrush>
</Border>
</Grid>
</ControlTemplate>
```

```

        </Setter.Value>
    </Setter>
</Style>
</Canvas.Resources>
<Slider Canvas.Left="10" Canvas.Top="30" Width="120" Height="25"/>
</Canvas>
</Page>

```



Figura 14-12 Slider que muestra sólo una barra

Pongamos ahora el Track con nombre PART_Track como lo indica la documentación de la clase Slider (Listado 14-9 y Figura 14-13).

Listado 14-9 Plantilla de Slider con Track

```

<Page ...>
<Canvas Background="LightSteelBlue">
<Canvas.Resources>
<Style TargetType="{x:Type Slider}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Slider}">
<Grid>
<Grid.RowDefinitions>
...
</Grid.RowDefinitions>
<Border ...>
...
</Grid>
<Track Grid.RowSpan="3" Name="PART_Track">
<Track.DecreaseRepeatButton>
<RepeatButton Command="Slider.DecreaseLarge" />
</Track.DecreaseRepeatButton>
<Track.IncreaseRepeatButton>
<RepeatButton Command="Slider.IncreaseLarge" />
</Track.IncreaseRepeatButton>
<Track.Thumb>
<Thumb/>
</Track.Thumb>
</Track>

```

```

    </ControlTemplate>
    </Setter.Value>
</Setter>
</Style>
</Canvas.Resources>
<Slider Canvas.Left="10" Canvas.Top="30" Width="120" Height="25"/>
</Canvas>
</Page>

```



Figura 14-13 Slider con Track

Vea este mismo Slider aumentado de tamaño 4 veces con ScaleTransform y observe que los dos RepeatButtons y el Thumb se muestran con sus plantillas predefinidas sobre el borde que dibuja la barra de rango. Como señalamos en un principio: cambiar la apariencia de un Slider se traduce en definir la apariencia del Thumb y los RepeatButtons.



Figura 14-14 Slider de la Figura 14-13 escalado 4 x 4

Hagamos pues tres estilos más, uno para el Thumb y uno para cada RepeatButton. Observe en la Figura 14-7 que el RepeatButton de la izquierda se muestra diferente al de la derecha para resaltar por dónde va el valor del Slider respecto del máximo. La Figura 14-15 muestra el Slider escalado y con la plantilla completa, y la Figura 14-16 nos muestra este Slider en el tamaño natural.

Listado 14-10 Plantilla final para el Slider de la aplicación de ejemplo del curso

```

<Page ...>
<Canvas Background="LightSteelBlue">
<Canvas.Resources>
<Style x:Key="SliderThumb" TargetType="{x:Type Thumb}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Thumb}">

```

```

<Rectangle RadiusX="4" RadiusY="2" Stroke="#4000"
    StrokeThickness="0.5" Width="10">
    <Rectangle.Fill>
        <LinearGradientBrush EndPoint="0,1">
            <GradientStop Color="#dfff" Offset="0"/>
            <GradientStop Color="#2000" Offset="1"/>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
<Style TargetType="{x:Type RepeatButton}" x:Key="SliderRepeatButton">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type RepeatButton}">
                <Border Background="Transparent" />
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
<Style TargetType="{x:Type RepeatButton}" x:Key="SliderFillRepeatButton">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type RepeatButton}">
                <Border Margin="1,1.5,0,1" CornerRadius="2,0,0,2">
                    <Border.Background>
                        <LinearGradientBrush EndPoint="0,1">
                            <GradientStop Color="Blue" Offset="0"/>
                            <GradientStop Color="CornflowerBlue" Offset="0.6"/>
                            <GradientStop Color="RoyalBlue" Offset="1"/>
                        </LinearGradientBrush>
                    </Border.Background>
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
<Style TargetType="{x:Type Slider}">
    <Setter Property="Template">
        <Setter.Value>

```

```

<ControlTemplate TargetType="{x:Type Slider}">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"
                MinHeight="{TemplateBinding MinHeight}"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Border Height="5" CornerRadius="2"
            Background="Transparent" BorderThickness="1">
            <Border.BorderBrush>
                <LinearGradientBrush EndPoint="0,1">
                    <GradientStop Color="#4000" Offset="0"/>
                    <GradientStop Color="#dfff" Offset="1"/>
                </LinearGradientBrush>
            </Border.BorderBrush>
        </Border>
        <Track Grid.RowSpan="3" Name="PART_Track" Grid.Column="1">
            <Track.DecreaseRepeatButton>
                <RepeatButton Style="{StaticResource SliderFillRepeatButton}"
                    Command="Slider.DecreaseLarge" />
            </Track.DecreaseRepeatButton>
            <Track.IncreaseRepeatButton>
                <RepeatButton Style="{StaticResource SliderRepeatButton}"
                    Command="Slider.IncreaseLarge" />
            </Track.IncreaseRepeatButton>
            <Track.Thumb>
                <Thumb Style="{StaticResource SliderThumb}" />
            </Track.Thumb>
        </Track>
    </Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Canvas.Resources>
<Slider Canvas.Left="10" Canvas.Top="30" Width="120" Height="25">
    <Slider.RenderTransform>
        <ScaleTransform ScaleX="4" ScaleY="4"/>
    </Slider.RenderTransform>
</Slider>
</Canvas>
</Page>

```



Figura 14-15 Slider de la Figura 14-16 ampliado 4 x 4



Figura 14-16 Slider con la plantilla final

Observe como cambiando sus estilos con `StaticResource` desde la definición de la plantilla del Slider hemos asignado también plantillas a los `RepeatButtons` y al `Thumb`.

Con este ejemplo cerramos esta lección de plantillas de controles pero no queremos terminar sin antes invitarle a que explore en el Windows SDK donde usted podrá encontrar variados ejemplos de estilos. Le recomendamos particularmente ver `ControlTemplateExamples` que puede encontrar en el directorio `Controls` de los ejemplos WPF del SDK. Examine los diversos estilos que usted puede ver definidos en la aplicación de ejemplo del curso, los cuales se aplican a los tipos `Button`, `Slider`, `DocumentViewer`, `TabControl` y hasta la propia Ventana (Window).

Definiendo sus propias plantillas Ud. podrá darle un sello personal a la apariencia de sus aplicaciones hasta donde sus habilidades de diseño le sean capaces de llevar. WPF le ayudará a ello.

Capítulo V ELEMENTOS AVANZADOS

En este capítulo se tratan temas más avanzados que le permitirán incluir en sus aplicaciones animaciones, audio, vídeo, gráficos 3D y documentos, algo que antes de WPF había que lograr con un trabajo bizarro y una amalgama de tecnologías

Lección 15 Animaciones

¿No se ha desesperado usted cuando el reloj de arena de algunas aplicaciones le parece que no cambia, o cuando la barra de progreso que indica el avance de una instalación le parece detenida? Si una aplicación permaneciese visualmente estática sería difícil discriminar si está funcionando bien o si se ha colgado. Si al hacer clic sobre un botón no se produjera un efecto visual puede que usted no se percate si ha hecho realmente el clic.

Darle animación a la interfaz de usuario de su aplicación es darle vida y mayor motivación al trabajo con la misma.

Usted posiblemente se haya ganado mayor atención del auditorio cuando le ha dado animación a una presentación preparada con Power Point. Si bien hasta ahora varias aplicaciones de Windows tienen animación (aunque el Clipo de Office no ha resultado simpático), lo cierto es que a los desarrolladores nos ha resultado difícil incorporarle animación a nuestras aplicaciones, salvo los efectos que ya viniesen predeterminados con algunos controles.

WPF facilita el añadirle animación a nuestras aplicaciones porque ofrece una infraestructura mediante la cual podemos animar casi cualquier aspecto visible de la interfaz de usuario. Usted podrá lograr efectos como los de Power Point y muchos más. Sobre animaciones es que trata esta lección.

15.1 ¿Qué entendemos por animación?

Una animación no es más que una secuencia de cuadros que al realizar transiciones entre un cuadro y otro darán un efecto visual de movimiento.

¿Quién de chico no hizo su pequeña animación en las páginas de su cuaderno? (Figura 15-1).

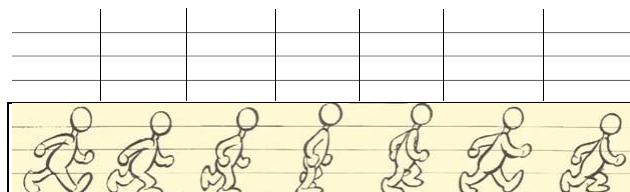


Figura 15-1 Mi primera animación

En términos de programación se puede lograr una animación con una secuencia de imágenes que se sucedan una tras otra en un intervalo de tiempo determinado.

Antes de WPF, los desarrolladores de aplicaciones Windows tenían que crear sus propios sistemas para manejar el tiempo de una animación o hacer uso de algunas bibliotecas especializadas. WPF incluye un eficiente sistema de manejo de tiempo que puede usarse a través de código .NET y XAML. Con las animaciones de WPF resulta relativamente fácil animar controles y otros elementos de interfaz de usuario con la comodidad de que es WPF quien se encarga de todo el trabajo de administrar el tiempo y hacer el "repintado" eficiente de la pantalla, necesarios para lograr un efecto realista con la animación. Con las clases que ofrece WPF Ud. se puede abstraer de cómo se logran los efectos, y se puede concentrar en cómo usarlos para lograr las animaciones.

El tipo de elemento más relacionado con las animaciones es Storyboard. Un StoryBoard es un contenedor de objetos de tipo Timeline (Todas las animaciones heredan de Timeline). Los objetos StoryBoard permiten combinar distintas animaciones que influyen sobre una variedad de objetos y propiedades agrupándolas como su contenido, haciendo más sencillos el control y organización del comportamiento de una animación en el tiempo. Todas las animaciones que se definen en el código XAML deben estar contenidas en un StoryBoard.

Los tipos de animaciones definidos en WPF heredan de la clase Timeline. Un Timeline define un segmento de tiempo. Con un Timeline se especifica la duración de la animación, cuantas veces se repetirá y la rapidez con que ocurren los cambios que define la animación en la duración que se le ha asignado.

La base de una animación consiste en ir generando valores que al asignárselos a propiedades estos cambios provoquen los efectos buscados (por ejemplo una transición de colores durante un intervalo de tiempo). De este modo para animar una propiedad que toma como valor un tipo Double, como es el caso del la propiedad Width de un elemento, se debe usar una animación que produzca valores de tipo Double. Del mismo modo para animar una propiedad que tome como valor un tipo T debe usarse una animación que genere valores de tipo T. Podemos por ejemplo hacer crecer o decrecer un elemento animando sus propiedades Height y Width como se ilustra en el Listado 15-1 a continuación (un código que ya le mostramos en la **Lección Triggers**), o hacer que un elemento se vaya haciendo visible paulatinamente en el interfaz de la aplicación animando su propiedad Opacity.

Listado 15-1 Código XAML, animando la altura de una imagen

```
<Window x:Class=" EjemploAnimaciones.Window1"  
  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title=" EjemploAnimaciones: EventTrigger" Height="300"
```

```

Width="300"
Name="Ventana" Background="DarkBlue" >
<Window.Resources>
<Style TargetType="{x:Type Image}">
<Style.Triggers>
<EventTrigger RoutedEvent="Image.MouseEnter">
<EventTrigger.Actions>
<BeginStoryboard>
<Storyboard>
<DoubleAnimation To="300" Duration="0:0:2"
Storyboard.TargetProperty="(Image.Height)" />
</Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Image.MouseLeave">
<EventTrigger.Actions>
<BeginStoryboard>
<Storyboard>
<DoubleAnimation Duration="0:0:2"
Storyboard.TargetProperty="(Image.Height)" />
</Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Style.Triggers>
</Style>
</Window.Resources>
<Image Height="100" HorizontalAlignment="Center"
VerticalAlignment="Center" Source="Windito.gif"/>
</Window>

```

En el ejemplo hemos definido la animación en un estilo para tipos Image, de esta forma para cualquier imagen a la que se aplique este estilo, la animación ocurrirá como ilustra la Figura 15-2.

La Figura 15-2 ilustra una secuencia del efecto "zoom" que provoca la animación cuando el cursor del ratón entra en el área de la imagen. Note que se ha programado también una animación en sentido inverso hasta que el Height retome su valor.



Figura 15-2 Animando la altura de una imagen

Para poder animar una propiedad, ésta debe ser una *dependency property* y debe ser de un tipo heredero de *DependencyObject* que brinde una implementación para la interfaz *IAnimable*. La mayoría de los elementos visuales tienen "propiedades animables", controles como los botones (*Button*), los paneles (*Panel*), las figuras (*Shape*) heredan de *DependencyObject* y la mayoría de sus propiedades son *dependency properties*. Las animaciones pueden ser usadas casi en cualquier lugar de una aplicación WPF, entiéndase estilos, plantillas, etc.

El comportamiento de una animación en el tiempo

En el tipo *TimeLine* existen un conjunto de propiedades relacionadas con el comportamiento de una animación. Las más usadas son *Duration*, *AutoReverse*, y *RepeatBehavior*. Con la propiedad *Duration* se especifica el tiempo que dura una animación. El tiempo es especificado con una cadena que tiene el formato *hh:mm:ss*, donde *hh* son las horas, *mm* los minutos y *ss* los segundos.

Con la propiedad booleana *AutoReverse* (que tiene valor *false* por defecto) podemos definir (si le damos valor *true*) que una vez concluida la animación ocurra de forma automática una suerte de "animación en sentido inverso" (algo así como un "*undo*" a las modificaciones provocadas por la animación) retornando el valor de la propiedad animada a su valor inicial no animado. Por ejemplo, si para la siguiente animación (extraída del Listado 15-1), se hace *AutoReverse="True"* se provocaría un comportamiento para el valor de la propiedad *Height* como el que se ilustra en la Figura 15-3 (La flecha en rojo indica el cambio de valor de la propiedad *Height* a 300 y la azul indica el proceso inverso desencadenado al asignar valor "**True**" a la propiedad **AutoReverse**):

```
<DoubleAnimation To="300" Duration="0:0:2"
    Storyboard.TargetProperty="(Image.Height)"
    AutoReverse="True"/>
```

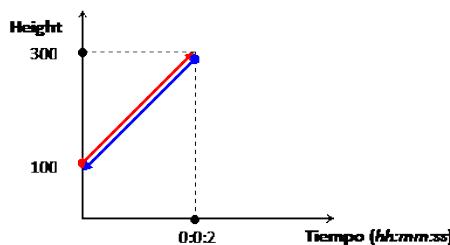


Figura 15-3 AutoReverse="True"

Con la propiedad RepeatBehavior (que tiene valor por defecto 1) se especifica cómo la animación se repite una vez culminado el segmento de tiempo asignado. Por ejemplo para una animación en que la propiedad Duration tenga valor "0:0:2"

Si hacemos RepeatBehavior="Forever" estamos diciendo que la animación se vuelve a repetir indefinidamente. Ver Figura 15-4 a continuación.

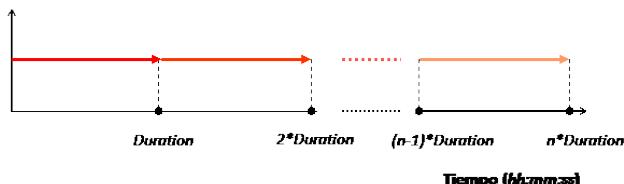


Figura 15-4 RepeatBehavior = Forever

Si hacemos RepeatBehavior="0:0:5" estamos diciendo que la animación se repite por cinco segundos. Como en este ejemplo la duración de cada animación se ha definido de 2 segundos, entonces significa que se repetirá dos veces y hasta donde alcance el minuto restante. Ver Figura 15-5 a continuación.

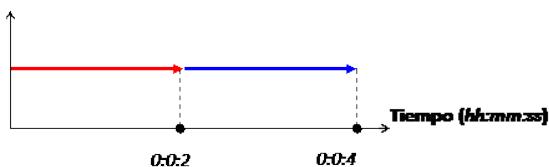


Figura 15- 5 RepeatBehavior="0:0:5"

Si hacemos RepeatBehavior="3x" estamos diciendo que la animación se repite tres veces. Ver Figura 15-6 a continuación.

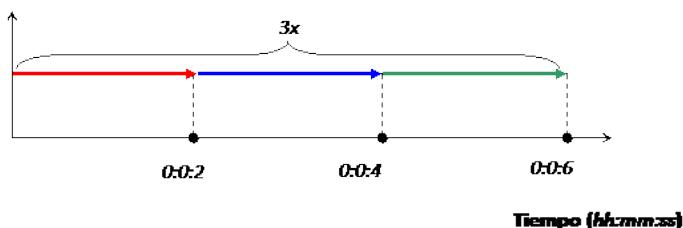


Figura 15- 6 RepeatBehavior="3x"

Otra propiedad que puede resultarle de gran utilidad a la hora de combinar varias animaciones en una línea de tiempo es BeginTime. Con BeginTime definimos el momento en que debe comenzar una animación, permitiendo con esto lograr continuidad y superposición entre distintas animaciones (Figura 15-7).

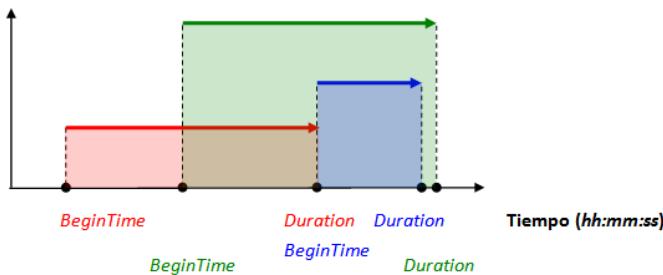


Figura 15-7 BeginTime

Podemos también hacer que la velocidad de una animación aumente o disminuya en el transcurso del tiempo. Predeterminadamente una animación ocurre a una velocidad constante que puede especificarse con la propiedad SpeedRatio (con la cual se puede indicar mayor o menor velocidad pero siempre constante). Para acelerar o desacelerar una animación contamos con las propiedades AccelerationRatio y DecelerationRatio.

Con estas propiedades especificamos en qué por ciento de la animación ocurre una aceleración o desaceleración. AccelerationRatio representa un por ciento del segmento de tiempo desde su valor inicial hasta el final y DecelerationRatio por su parte representa un por ciento del segmento de tiempo desde su final hasta el valor inicial. A estas propiedades se le debe asignar valor entre 0 y 1 y la suma de los valores de ambas no debe excederse de 1 (el 100%) que representa al tiempo de la total de la animación (Duration). Por defecto el valor de las propiedades AccelerationRatio y DecelerationRatio es 0.

Por ejemplo para expresar que la velocidad de una animación acelere durante el primer 40% del tiempo de duración y comience a desacelerar en el último 30% del tiempo total hasta finalizar, se debe escribir:

AccelerationRatio="0.4" DecelerationRatio="0.3".

En este caso esto significa que cuando transcurra el 40% del tiempo seguirá con la velocidad que había alcanzado y cuando llegue al 70% comenzará a desacelerar. Ver Figura 15-8 a continuación.

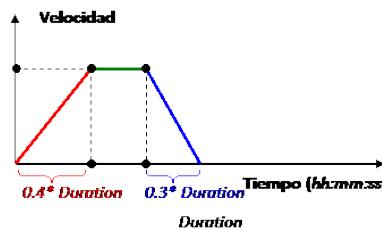


Figura 15-8 AccelerationRatio="0.4" DecelerationRatio="0.3"

Existen otras propiedades para las animaciones igual de interesantes, pero no es objetivo de este curso explicarlas todas en detalle. Lo invitamos a continuar con la lección y luego a experimentar por su cuenta.

15.2 Tipos de animaciones

A la par de la gran variedad de tipos de valores que puede tomar una propiedad, existe una amplia variedad de tipos de animaciones definidos en el espacio de nombres System.Windows.Media.Animations. Afortunadamente, las animaciones siguen una estricta convención de nombres que facilita diferenciarlas.

Para ilustrarle el uso de las animaciones, nos apoyaremos en el siguiente ejemplo (Listado 15-2) que nos pone un balón en el campo de fútbol e iremos realizando los distintos tipos de animaciones sobre el balón (Image que aparece en el código).

Listado 15-2 Código XAML, Balón sin animar

```
<Window x:Class="EjemploAnimaciones.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="EjemploAnimaciones"           Height="557"           Width="726"
Background="Green"  >
    <Canvas Height="300" Width="300"
        HorizontalAlignment="Left" VerticalAlignment="Top">
        <Image Source="Campo.jpg"/>
        <Image Source="Balón.gif" Name="Balón" Width="100"
            Canvas.Top="300" Canvas.Left="10"></Image>
    </Canvas></Window>
```

La Figura 15-9 ilustra el resultado de la ejecución del código XAML del Listado 15-2.

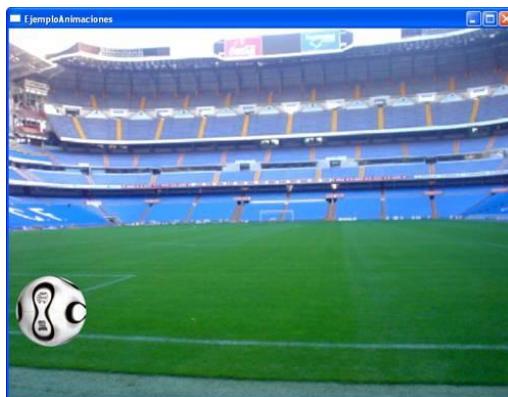


Figura 15-9 Balón sin animar.

15.3 Animaciones From/To/By (haciendo un pase raso del balón)

Los nombres de las animaciones From/To/By se ajustan a la siguiente convención de nombres:

<Tipo>Animation

Donde <Tipo> es el nombre del tipo de la propiedad que se desea animar.

Esta animación conocida como animación "From/To/By" se considera la animación básica, es una animación que ocurre entre dos valores inicio y final y va incrementando con un valor de incremento el valor de inicio. De este modo con este tipo de animaciones es posible generar una transición entre un valor inicial y un valor final en el intervalo de tiempo en que ocurre la animación especificado mediante la propiedad Duration.

Tiene una propiedad From con la que se especifica el valor inicio y una propiedad To para especificar el valor final.

En lugar de la propiedad To se puede usar una propiedad By con la que se indica que el valor final para la animación será igual a la suma del inicial más el que se le asigne a la propiedad By. Las propiedades To y By son excluyentes, en caso de que explícitamente se les de valor a ambas, By será ignorado.

Veamos algunas combinaciones de estas propiedades:

From="50" To="100" el valor de la propiedad que se esté animando cambiará desde 50 hasta 100, con un incremento distribuido proporcionalmente según la duración de la animación.

From="50" By="100" el valor de la propiedad que se esté animando cambiará desde 50 hasta 150.

Si se escribe solo From="50" en este caso el valor de To se considera el predeterminado por lo que la propiedad que se esté animando cambiará desde 50 hasta su valor por defecto.

Puede parecer redundante la variante By porque hacer From="50" By="100" sería lo mismo que hacer From="50" To="150". La utilidad en tener también la variante From By radica en que no es necesario conocer el valor que se le da a From para asignar a la propiedad que se desea animar (este es el caso por ejemplo cuando el valor del To depende de un enlace a datos y no se conoce estáticamente cuando se escribe la animación). Recuerde que en XAML no es posible escribir expresiones para asignar como valor a una propiedad, es decir no es posible escribir algo como From="*enlace*" To="*enlace* + 100".

Apliquemos entonces una animación From To al balón del Listado 15-2 de la forma que se ilustra en el Listado 15-3 y empezaremos a "mover el balón"

Listado 15-3 Código XAML, haciendo un pase raso

```

<Image Source="Balón.gif" Name="Balón" Width="100"
    Canvas.Top="350" Canvas.Left="10">
    <Image.RenderTransform>
        <TranslateTransform x:Name="TTranslate"/>
    </Image.RenderTransform>
    <Image.Triggers>
        <EventTrigger RoutedEvent="Image.MouseLeftButtonDown">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Storyboard.TargetName="TTranslate"
                            Storyboard.TargetProperty="X"
                            From="0" To="620" Duration="0:0:10"/>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </Image.Triggers>
</Image>

```

Note en el Listado 15-3 cómo se ha animado el valor de la propiedad X de la transformación TranslateTransform (a la que se le dio nombre TTranslate) que se aplica a la imagen. Esto provocaría un resultado como el que se muestra en la Figura 15-10 de trasladar el balón en el sentido horizontal, la flecha amarilla representa la trayectoria que sigue la imagen una vez aplicada la animación.

En las figuras de esta lección tratamos de expresar solo "algunos cuadros" de la animación, ya que es imposible mostrarle la animación en este documento. Para una mejor comprensión ejecute los códigos de ejemplo de esta lección, o vea el video asociado. Para que se vean mejor las secuencias hemos hecho que el fondo de la ventana se muestre en color negro.

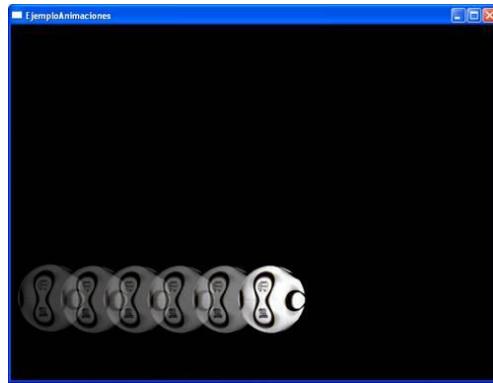


Figura 15-10 Un pase raso.

Por supuesto que esta animación no resulta del todo realista, el balón se ha trasladado pero estamos viendo siempre una misma cara del mismo porque no ha rotado. Para darle más realismo agregamos otra animación para hacer que el balón rote y a la vez que se desplace durante el tiempo que dure la animación. Esto es lo que hace el Listado 15-4. Note como se han combinado dos animaciones distintas en un mismo intervalo de tiempo. Hemos definido `CenterX="50" CenterY="50"` para la transformación `RotateTramsform` para lograr el efecto del que el balón rote sobre su centro (vea **Lección 10 Transformaciones**).

Listado 15-4 Código XAML, rotando el balón

```
<Image Source="Balón.gif" Name="Balón" Width="100"
       Canvas.Top="350" Canvas.Left="10">
  <Image.RenderTransform>
    <TransformGroup>
      <RotateTransform           CenterX="50"           CenterY="50"
x:Name="TRotate"/>
      <TranslateTransform x:Name="TTranslate"/>
    </TransformGroup>
  </Image.RenderTransform>
  <Image.Triggers>
    <EventTrigger RoutedEvent="Image.MouseLeftButtonDown">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation Storyboard.TargetName="TRotate"
                             Storyboard.TargetProperty="Angle"
                             From="0" To="360" Duration="0:0:10"/>
            <DoubleAnimation Storyboard.TargetName="TTranslate"
                             Storyboard.TargetProperty="X"
                             From="0" To="620" Duration="0:0:10"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Image.Triggers>
</Image>
```

```

</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Image.Triggers>
</Image>

```

Note en el Listado 15-4 como se accede a cada una de las transformaciones a través del nombre que se le ha asignado a cada transformación (en este caso TRotate y TTranslate). Se han animado los valores de las propiedades X de TranslateTransform y Angle de RotateTransform. Esto provocaría un resultado como el que se muestra en la Figura 15-11 (la línea punteada en amarillo representa la traslación y la naranja la rotación).



Figura 15-11 Un pase raso (el balón rota)

Este recurso de las líneas punteadas pretende ilustrarle en una figura lo que ocurriría durante la animación. Para apreciar verdaderamente el efecto ejecute los proyectos que acompañan a esta lección.

La tabla a continuación muestra algunos de los tipos de animaciones From/To/By más usados.

Tipo de la propiedad	Animación From/To/By
Color	ColorAnimation
Double	DoubleAnimation
Point	PointAnimation
String	

15.4 Animaciones Key Frame (rebotando el balón)

En WPF, existe también un grupo de animaciones conocidas como animaciones por cuadros (Key-Frame). Una animación por cuadros ocurre a lo largo de cualquier número de valores, descritos por objetos "cuadros" (implementan la interface IKeyFrame) almacenados en una colección como contenido de la animación.

Con una animación From/To/By no se pueden especificar más de dos valores para la ocurrencia de la animación en un intervalo de tiempo. Si se quiere que una animación contenga más de dos valores objetivos se debe usar una animación por cuadros.

De igual forma que las animaciones From/To/By, una animación por cuadros modificará el valor de una determinada propiedad al ir transcurriendo el tiempo. De cualquier modo, mientras que con una animación From/To/By se establece una transición entre dos valores, con una animación por cuadros se pueden crear transiciones entre varios valores en intervalos del tiempo distintos dentro del transcurso de la animación (Duration). Una animación por cuadros se define mediante un conjunto de "cuadros" (de ahí el término, "key-frame animation"). Para especificar los valores de la animación, se crea un objeto "cuadro" y se agrega a la colección de KeyFrames de la animación y cuando ésta ejecute, WPF realizará una transición entre los cuadros especificados.

El principio de cualquier animación, consiste en tener un conjunto de cuadros principales dentro del flujo de una escena. La animación resultará de una interpolación entre un cuadro principal y el que le sigue (Figura 15-12), por supuesto que mientras más cuadros conformen la animación más "suave" serán las transiciones entre estos, y mas realista se verá la aplicación

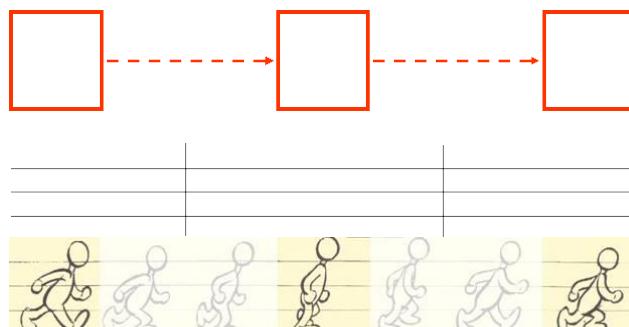


Figura 15- 12 Cuadros principales de una animación.

Una animación por cuadros permite establecer una transición entre dos o más valores. Existen distintos tipos de animaciones por cuadros para animar los diferentes tipos de propiedades con que contamos. De igual forma existen también distintos tipos de objetos "cuadros": uno por cada tipo de valor animado y método de interpolación que éste aplica. Afortunadamente los nombres de las animaciones por cuadros se ajustan a la siguiente convención de nombres:

<Método de interpolación><Tipo>KeyFrame

Donde <Método de interpolación> es el método usado para hacer la transición entre un cuadro y otro de la animación y <Tipo> es el tipo de la propiedad que se desea animar, por ejemplo DiscreteDoubleKeyFrame.

Una animación por cuadros con soporte para los tres métodos de interpolación en WPF, debe brindar tres tipos "cuadros". Por ejemplo, Ud. puede usar en una animación los tres tipos de cuadro: DiscreteDoubleKeyFrame, LinearDoubleKeyFrame y SplineDoubleKeyFrame.

Con las animaciones por cuadros (**Key-Frame**) se puede especificar cualquier cantidad de intervalos para los valores a animar, a diferencia de las **From/To/By** donde los valores de la animación cambian dentro de un solo intervalo (definido por un inicio-final), así como controlar el método de interpolación para la transición entre estos valores.

Vea en el Listado 15-5 el uso de `LinearDoubleKeyFrame`. Note que como queremos realizar una animación entre distintos valores (cada uno de los cambios de trayectoria del balón representado con los cuadros en la Figura 15-13) no sería factible el uso de una animación **From/To/By**.

Listado 15-5 Código XAML, un pase por cuadros (Key-Frame)

```
<Image Source="Balón.gif" Name="Balón" Width="100"
    Canvas.Top="350" Canvas.Left="10">
    <Image.RenderTransform>
        <TransformGroup>
            <RotateTransform CenterX="50" CenterY="50" x:Name="TRotate"/>
            <TranslateTransform x:Name="TTranslate"/>
        </TransformGroup>
    </Image.RenderTransform>
    <Image.Triggers>
        <EventTrigger RoutedEvent="Image.MouseLeftButtonDown">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetName="TRotate"
                            Storyboard.TargetProperty="Angle"
                            From="0" To="360" Duration="0:0:10"/>
                        <DoubleAnimation Storyboard.TargetName="TTranslate"
                            Storyboard.TargetProperty="X"
                            From="0" To="620" Duration="0:0:10"/>
                        <DoubleAnimationUsingKeyFrames
                            Storyboard.TargetName="TTranslate"
                            Storyboard.TargetProperty="Y"
                            Duration="0:0:10">
                            <LinearDoubleKeyFrame Value="-200" KeyTime="0:0:1"/>
                            <LinearDoubleKeyFrame Value="0" KeyTime="0:0:2"/>
                            <LinearDoubleKeyFrame Value="-150" KeyTime="0:0:3"/>
                        
```

```

<LinearDoubleKeyFrame Value="0" KeyTime="0:0:4"/>
<LinearDoubleKeyFrame Value="-100" KeyTime="0:0:5"/>
<LinearDoubleKeyFrame Value="0" KeyTime="0:0:6"/>
<LinearDoubleKeyFrame Value="-50" KeyTime="0:0:7"/>
<LinearDoubleKeyFrame Value="0" KeyTime="0:0:8"/>
<LinearDoubleKeyFrame Value="-25" KeyTime="0:0:9"/>
<LinearDoubleKeyFrame Value="0" KeyTime="0:0:10"/>
</DoubleAnimationUsingKeyFrames>
</Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Image.Triggers>
</Image>

```

Si el valor de la propiedad Duration es Automatic o igual al valor de tiempo del último cuadro, la animación termina. De otro modo, si Duration es superior al tiempo del último cuadro (KeyTime), la animación mantiene el valor del cuadro hasta alcanzar el fin (Duration), todas las animaciones hace uso de la propiedad FillBehavior para saber determinar si mantener su valor final cuando alcance el final del período activo de la animación.



Figura 15- 13 El balón dando botes por “cuadros”.

Formas de interpolación

Además de soportar múltiples valores, algunas animaciones soportan distintas formas de interpolación. La interpolación de una animación define como ocurre la transición desde un valor a otro (líneas punteadas en la Figura 15-13). Hay tres tipos de interpolaciones:

- Interpolación Lineal (**Linear**)

Con una interpolación lineal, la animación progresó de forma constante en el tiempo de duración.

- Interpolación Discreta (**Discrete**)

Con una interpolación discreta, la función de animación salta de un valor a otro sin realizar interpolación.

- Interpolación con Spline (**Spline**)

Realiza una interpolación *Spline* entre los cuadros de la animación. La interpolación mediante *Spline* puede usarse para lograr escenarios más realistas. Permite controlar la aceleración o desaceleración de los objetos, así como un detallado manejo de los segmentos del tiempo, permitiendo simular efectos de mundo real.

Una curva cúbica *Bezier* es definida por un punto de comienzo, uno final y dos puntos de control. La propiedad KeySpline permite definir estos dos puntos de control de una curva *Bezier* que se extiende desde el punto $(0,0)$ al $(1,1)$. El primer punto de control, controla el factor de curvatura en la primera mitad de la curva *Bezier* y el segundo en la segunda mitad. La curva resultante describe el ritmo de cambios para un cuadro en una animación con interpolación *Spline*. Mientras más empinada sea la curva más rápido el cuadro cambia sus valores. A medida que disminuye la curvatura los cambios se hacen más lentos (Figura 15-14).

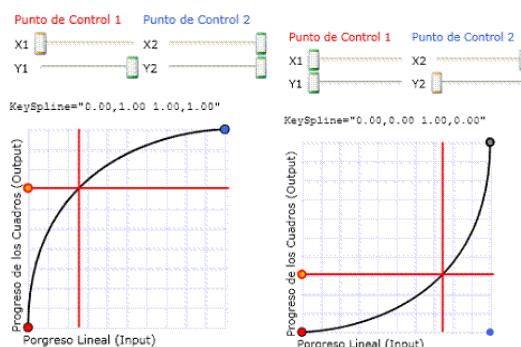


Figura 15- 14 Izquierda (en 1 más velocidad). Derecha (En 2 más velocidad).

En el Listado 15-5 utilizamos una interpolación lineal, en este caso la trayectoria para lograr el efecto de ascenso y descenso del balón se ve muy recta. Veamos ahora como haciendo uso de una interpolación por *Spline* podemos suavizar la transición entre los cuadros, para dar mayorrealismo a la animación del balón dando un efecto de desaceleración al subir y aceleración al bajar, equivalente al que ocurre producto de la acción de la fuerza de gravedad (Listado 15-6). Para ello tomaremos como puntos de control $(0,1)$ y $(1,1)$ para la primera parte de la trayectoria

del balón (la subida) y $(0,0)$ y $(1,0)$ para la segunda trayectoria (la caída) obteniendo curvas de Bezier similares las de la Figura 15-14.

Listado15-6 Código XAML, suavizando el pase con un SplineDoubleKeyFrame

```
<Image Source="Balón.gif" Name="Balón" Width="100"
    Canvas.Top="350" Canvas.Left="10">
    <Image.RenderTransform>
        <TransformGroup>
            <RotateTransform CenterX="50" CenterY="50"
                x:Name="TRotate"/>
            <TranslateTransform x:Name="TTranslate"/>
        </TransformGroup>
    </Image.RenderTransform>
    <Image.Triggers>
        <EventTrigger RoutedEvent="Image.MouseLeftButtonDown">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetName="TRotate"
                            Storyboard.TargetProperty="Angle"
                            From="0" To="360" Duration="0:0:10"/>
                        <DoubleAnimation Storyboard.TargetName="TTranslate"
                            Storyboard.TargetProperty="X"
                            From="0" To="620" Duration="0:0:10"/>

                        <DoubleAnimationUsingKeyFrames
                            Storyboard.TargetName="TTranslate"
                            Storyboard.TargetProperty="Y"
                            Duration="0:0:2">
                            <SplineDoubleKeyFrame KeySpline="0.00,1.00
                                1.00,1.00"
                                Value="-200" KeyTime="0:0:1"/>
                            <SplineDoubleKeyFrame KeySpline="0.00,0.00
                                1.00,0.00"
                                Value="0" KeyTime="0:0:2"/>
                            <SplineDoubleKeyFrame KeySpline="0.00,1.00
                                1.00,1.00"
                                Value="-150" KeyTime="0:0:3"/>
                            <SplineDoubleKeyFrame KeySpline="0.00,0.00
                                1.00,0.00"
                                Value="0" KeyTime="0:0:4"/>
                        </DoubleAnimationUsingKeyFrames>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </Image.Triggers>

```

```

1.00,0.00"
    Value="0" KeyTime="0:0:4"/>
    <SplineDoubleKeyFrame    KeySpline=" 0.00,1.00
1.00,1.00"
    Value="-100" KeyTime="0:0:5"/>
    <SplineDoubleKeyFrame    KeySpline=" 0.00,0.00
1.00,0.00"
    Value="0" KeyTime="0:0:6"/>
    <SplineDoubleKeyFrame    KeySpline=" 0.00,1.00
1.00,1.00"
    Value="-50" KeyTime="0:0:7"/>
    <SplineDoubleKeyFrame    KeySpline=" 0.00,0.00
1.00,0.00"
    Value="0" KeyTime="0:0:8"/>
    <SplineDoubleKeyFrame    KeySpline=" 0.00,1.00
1.00,1.00"
    Value="-25" KeyTime="0:0:9"/>
    <SplineDoubleKeyFrame    KeySpline=" 0.00,0.00
1.00,0.00"
    Value="0" KeyTime="0:0:10"/>
</DoubleAnimationUsingKeyFrames>
</Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Image.Triggers>
</Image>

```

Vea ahora como ocurre una trayectoria más suave, el efecto visual que se obtiene es similar al que se ha representado con las líneas punteadas curvas amarillas en la Figura 15-15. Para observar los efectos de aceleración y desaceleración ejecute los códigos de ejemplo que acompañan a esta lección.



Figura 15-15 Cuadros principales de una animación Key Frame(Interpolación Spline).

La tabla a continuación muestra algunos de los tipos de animaciones a cuadros más usados.

Tipo de la propiedad	Animaciones Key-frame
Color	ColorAnimationUsingKeyFrames
Double	DoubleAnimationUsingKeyFrame
Point	PointAnimationUsingKeyFrame
String	StringAnimationUsingKeyFrame

15.5 AnimationPath (el pase de un crack)

Es posible realizar animaciones que se rigen por un determinado path geométrico (Vea **Lección Figuras**). Las animaciones "path" se rigen por la siguiente convención de nombres:

<Tipo>AnimationUsingPath

Donde <Tipo> es nombre del tipo del valor que se desea animar.

Todas estas animaciones tienen la propiedad PathGeometry, que es una colección de tipos PathFigure (Vea **Lección Figuras**). La animación genera sus salidas basándose en este PathGeometry.

Hay tres clases de esta forma de animación, estas son DoubleAnimationUsingPath, PointAnimationUsingPath, y MatrixAnimationUsingPath.

DoubleAnimationUsingPath genera valores de tipo Double a partir del PathGeometry asociado. Con la propiedad Source, se especifica que DoubleAnimationUsingPath use la coordenada **x**, la coordenada **y**,o el ángulo del path, como salida (output) para la animación

PointAnimationUsingPath genera valores de tipo Point a partir de las coordenadas **x** y **y** de su PathGeometry. No se pueden rotar objetos con esta animación.

MatrixAnimationUsingPath genera valores de tipo Matrix a partir de su PathGeometry. Cuando se aplica un MatrixTransform, entonces un MatrixAnimationUsingPath puede mover un elemento a través de un "path". Si se le asigna true a la propiedad DoesRotateWithTangent de MatrixAnimationUsingPath, entonces además el objeto rotará a lo largo de las curvas del path.

Vea por ejemplo (Figura 15-16) como un rectángulo puede ser animado con un path, note como el rectángulo avanza por las coordenadas que definen el path y como su ángulo de inclinación cambia en correspondencia con el ángulo de inclinación del path.

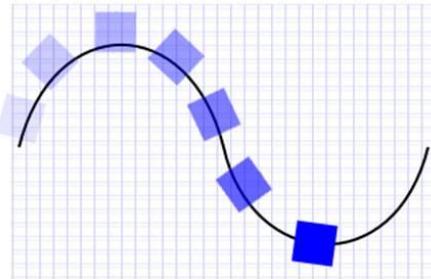


Figura 15-16 Rectángulo animado con una animación path.

Veamos ahora un ejemplo de animación usando un QuadraticBezierSegment (Listado 15-7) y un ArcSegment (Listado 15-8). Le invitamos nuevamente a remitirse a la **Lección 7 Figuras** y tratar de aplicar a las animaciones algunos de los tipos PathGeometry que allí se le mostraron. Como Ud. notará el código del Listado 15-7 no termina, más bien continúa en el Listado 15-8. Hemos hecho dos animaciones "path" una a continuación de la otra y para lograr un mejor entendimiento hemos interrumpido el código mostrándole lo que cada una provocaría en la ejecución.

Listado 15-7 Código XAML, DoubleAnimationUsingPath (QuadraticBezierSegment)
<pre><Image Source="Balón.gif" Name="Balón" Width="100" Canvas.Top="350" Canvas.Left="10"> <Image.RenderTransform> <TransformGroup> <RotateTransform CenterX="50" CenterY="50" x:Name="TRotate"/> <TranslateTransform x:Name="TTranslate"/> </TransformGroup> </Image.RenderTransform> <Image.Triggers> <EventTrigger RoutedEvent="Window.Loaded"> <EventTrigger.Actions> <BeginStoryboard></pre>

```

<Storyboard>
    <ParallelTimeline BeginTime="0:0:3">
        <DoubleAnimation Storyboard.TargetName="TRotate"
Storyboard.TargetProperty="Angle"           From="0"           To="360"
Duration="0:0:2"/>
        <DoubleAnimationUsingPath
            Storyboard.TargetName="TTranslate"
            Storyboard.TargetProperty="X"
            Source="X" Duration="0:0:2">
            <DoubleAnimationUsingPath.PathGeometry>
                <PathGeometry>
                    <PathFigure>
                        <QuadraticBezierSegment Point1="50,-250"
                                              Point2="550,-250"/>
                    </PathFigure>
                </PathGeometry>
            </DoubleAnimationUsingPath.PathGeometry>
        </DoubleAnimationUsingPath>
        <DoubleAnimationUsingPath
            Storyboard.TargetName="TTranslate"
            Storyboard.TargetProperty="Y"
            Source="Y" Duration="0:0:2">
            <DoubleAnimationUsingPath.PathGeometry>
                <PathGeometry>
                    <PathFigure>
                        <QuadraticBezierSegment Point1="50,-250"
                                              Point2="550,-250"/>
                    </PathFigure>
                </PathGeometry>
            </DoubleAnimationUsingPath.PathGeometry>
        </DoubleAnimationUsingPath>
    </ParallelTimeline>

```

Vea en la línea amarilla la trayectoria que sigue el balón (Figura 15-17), descrita por un QuadraticBezierSegment.



Figura 15-17 Haciendo un pase con un Path geométrico (QuadraticBezierSegment)

Veamos ahora la otra animación que se describe en el código XAML del Listado 15-8 (que completa el código del Listado 15-7) haciendo uso de un ArcSegment para ilustrar la trayectoria final del balón luego de que nuestra mascota diera un buen testarazo. Hemos omitido parte del código necesario para obtener el resultado que Ud. puede ver en la Figura 15-17, para no desviar la atención del objetivo de presentarle las animaciones "path". De cualquier manera Ud. puede remitirse a los códigos de ejemplo o al video asociado a esta lección para ver el ejemplo completo.

Listado 15-8 Código XAML, DoubleAnimationUsingPath (ArcSegment)

```
<DoubleAnimation BeginTime="0:0:2"
    Storyboard.TargetName="TRotate"
    Storyboard.TargetProperty="Angle"
    To="-360" Duration="0:0:2"/>
<DoubleAnimationUsingPath BeginTime="0:0:2"
    Storyboard.TargetName="TTranslate"
    Storyboard.TargetProperty="X"
    Source="X" Duration="0:0:2">
    <DoubleAnimationUsingPath.PathGeometry>
        <PathGeometry>
            <PathFigure StartPoint="300,-200">
                <ArcSegment Point="10,0" Size="300,300"/>
            </PathFigure>
        </PathGeometry>
    </DoubleAnimationUsingPath.PathGeometry>
</DoubleAnimationUsingPath>
<DoubleAnimationUsingPath BeginTime="0:0:2"
    Storyboard.TargetName="TTranslate"
```

```

        Storyboard.TargetProperty="Y"
        Source="Y" Duration="0:0:2">
    <DoubleAnimationUsingPath.PathGeometry>
        <PathGeometry>
            <PathFigure StartPoint="300,-200">
                <ArcSegment Point="10,0" Size="300,300"/>
            </PathFigure>
        </PathGeometry>
    </DoubleAnimationUsingPath.PathGeometry>
</DoubleAnimationUsingPath>
</ParallelTimeline>
</Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Image.Triggers>
</Image>

```

El resultado de este código sería la trayectoria que se ilustra en la Figura 15-18 a continuación



Figura 15-18 Dando un cabezazo con un Path geométrico (ArcSegment)

Con las animaciones basadas en un "path", en lugar de definir valores para las propiedades al estilo From/To/By o usar cuadros, definiremos un "path" geométrico para asignarle a la propiedad PathGeometry. A medida que la animación progresá, son tomados los valores **x** y **y**, así como información del ángulo en el "path" para generar la salida (output), logrando un efecto de movimiento.

Como habrá apreciado en esta lección, con las animaciones podemos imprimir un mayor dinamismo a la interfaz de nuestras aplicaciones, podemos hacer que los controles se muevan

por la ventana y que los textos aparezcan y desaparezcan. También podemos lograr una serie de divertidos efectos visuales. Hemos estudiado aquí las bases de WPF y XAML para hacer animaciones, por lo que ya puede usted dar rienda suelta a su imaginación para mejorar la apariencia e interactividad de sus aplicaciones.

Lección 16 Media

Tradicionalmente en las interfaces de usuario se han utilizado recursos gráficos como los controles y las imágenes. Últimamente se han impuesto elementos gráficos adicionales como las figuras redondeadas, los gradientes de colores, efectos visuales como las sombras o los relieves, etc. En lecciones anteriores se ha estudiado cómo WPF nos ofrece todo esto y más.

¿Cuán interesante puede usted hacer su interfaz de usuario si ahora pudiera incorporar sonido y video con la misma facilidad con la que pone un botón?

16.1 Media Element

WPF nos brinda el elemento `MediaElement` que se puede usar desde XAML como hasta ahora hemos puesto cualquier otro `FrameworkElement`. Comenzaremos esta lección mostrando el ejemplo de **Bienvenido a WPF** con audio en vez de con un texto en pantalla (Listado 16-1). No hemos puesto para este ejemplo ninguna figura porque el resultado de la ejecución es sólo audio.

Listado 16-1 Uso de `MediaElement` para decir el mensaje Bienvenido a WPF

```
<Window x:Class="MediaSample.Window1"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="MediaSample" Height="300" Width="300" >
    <Grid>
        <MediaElement Source="Bienvenido a WPF.wma"/>
    </Grid>
</Window>
```

Si fuera a mostrarse un video lo que habría que hacer en el Listado 16-1 es darle como valor a la propiedad Source del MediaElement en nombre de un archivo de video en lugar de uno de audio. El Listado 16-2 muestra una ventana como la de la Figura 16-1.

Listado 16-2 Uso de MediaElement para mostrar un video

```
<Window x:Class="MediaSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MediaSample" Height="300" Width="300" >
    <Grid>
        <MediaElement Source="Bee.wmv" Margin="5" />
    </Grid>
</Window>
```

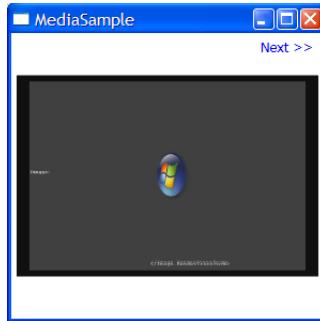


Figura 16-1 Resultado de ejecutar un MediaElement de un archivo de video

Observe que el MediaElement se pone en el panel con la misma facilidad con que usted pone otro elemento cualquiera como un rectángulo o un botón. Incluso como con cualquier otro FrameworkElement se le puede aplicar cualquier efecto visual o transformación. El Listado 16-3 nos mostraría el video rotado un ángulo y con efecto de hundimiento y suavizado como se muestra en la Figura 16-2.

Claro que para observar el efecto verdaderamente, usted debiera ejecutar el proyecto correspondiente que se adjunta a esta lección.

Listado 16-3 MediaElement con efectos de Bitmap y Transformación

```
<Window ...>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
```

```

</Grid.RowDefinitions>
<MediaElement Source="Media\bee.wmv" Margin="5">
    <MediaElement.BitmapEffect>
        <BitmapEffectGroup>
            <DropShadowBitmapEffect Color="#8000" Direction="315"
ShadowDepth="6"
                Softness="0.2"/>
            <BevelBitmapEffect Relief="0.3" Smoothness="0.3"
EdgeProfile="BulgedUp"
                BevelWidth="40" LightAngle="135"/>
        </BitmapEffectGroup>
    </MediaElement.BitmapEffect>
    <MediaElement.LayoutTransform>
        <SkewTransform AngleX="-5" AngleY="5"/>
    </MediaElement.LayoutTransform>
</MediaElement>
</Grid>
</Window>

```



Figura 16-2 MediaElement con efectos de bitmap y transformación de rotación

WPF reproduce una amplia variedad de formatos de audio y vídeo. En el caso de un archivo de audio WPF no muestra nada visual como resultado de ejecutar el MediaElement, es asunto del desarrollador incorporar algún elemento visual si quiere acompañar a la reproducción del sonido de alguna apariencia visual.

16.2 Operaciones de Reproducción

Si usted probara a ejecutar el código del Listado 16-1 y el Listado 16-2, notará que el mensaje de audio o el video comienzan a reproducirse por el MediaElement inmediatamente que la ventana aparece en pantalla. Un MediaElement tiene una propiedad LoadedBehavior de tipo MediaState que tiene valor por defecto Play de la propiedad. Si usted quisiera llevar el control de cuándo

comienza a reproducirse el video, detenerlo, echarlo a andar nuevamente o repetir la reproducción usted puede entonces indicar que el valor de LoadedBehavior es Manual y utilizar entonces los métodos Play, Pause o Stop del elemento MediaElement como se muestra en el Listado 16-4.

Listado 16-4 MediaElement controlado manualmente por botones de control de reproducción

```
<Window x:Class="MediaSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MediaSample" Height="300" Width="300" >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <MediaElement x:Name="media" Source="Media\bee.wmv" Margin="5"
        LoadedBehavior="Manual"/>
        <StackPanel Orientation="Horizontal" Grid.Row="1"
Background="LightSteelBlue">
            <Button Margin="5" Click="OnPlayClick" Width="50">Play</Button>
            <Button Margin="5" Click="OnPauseClick" Width="50">Pause</Button>
            <Button Margin="5" Click="OnStopClick" Width="50">Stop</Button>
        </StackPanel>
    </Grid>
</Window>
```

La Figura 16-3 muestra cómo se vería esta aplicación. Le recomendamos especialmente que para estos ejemplos pruebe usted mismo a ejecutar cada uno de los códigos que aquí mostramos para que pueda verdaderamente apreciar el audio y el vídeo. El código C# de los métodos OnPlayClick, OnPauseClick y OnStopClick son los que se ilustran en el Listado 16-5.



Figura 16-3 MediaElement y botones de control de reproducción

Listado 16-5 Uso de los métodos Play, Pause y Stop del tipo MediaElement

```
public partial class Window1 : System.Windows.Window {  
    public Window1() {  
        InitializeComponent();  
    }  
    private void OnPlayClick(object sender, EventArgs args) {  
        media.Play();  
    }  
    private void OnPauseClick(object sender, EventArgs args) {  
        media.Pause();  
    }  
    private void OnStopClick(object sender, EventArgs args) {  
        media.Stop();  
    }  
}
```

Usted puede emplear estas y otras operaciones para reproducir, detener, rebobinar y demás operaciones de video y audio utilizando elementos de tipo MediaElement que puede poner en cualquier parte de su aplicación WPF, sin embargo muchas de estas operaciones que hemos visto antes se hacen más fácil y a la vez se enriquecen mucho si se hicieran sólo desde XAML utilizando animaciones.

16.3 Animación con Audio y Video

Como los efectos de audio y vídeo tienen una duración de tiempo se pueden usar los elementos descritos en la Lección **Animaciones** para echar a andar una secuencia de sonido o video. Vamos a retomar el ejemplo de la sección anterior pero empleando esta vez animaciones para controlar la reproducción del audio o del video.

Todas las animaciones en WPF están regidas por líneas de tiempo (TimeLines). Con estas líneas de tiempo se puede controlar la duración de una animación, su velocidad y aceleración, el momento de inicio y demás posibilidades que se describen en la Lección **Animaciones**. WPF brinda una línea de tiempo particular para controlar la reproducción de audio o video: MediaTimeLine.

En el Listado 16-6 se puede ver un StoryBoard en los recursos de la ventana (Window.Resources) con un elemento de tipo MediaTimeLine donde hemos indicado un video en su propiedad Source y además hemos indicado como elemento destino de la animación

(Storyboard.TargetName) el nombre del MediaElement de esta aplicación, "media" en este caso.

Listado 16-6 StoryBoard que controla la reproducción de un video

```
<Window x:Class="MediaSample.MediaAsAnimation"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MediaSample" Height="300" Width="300" >
<Window.Resources>
    <Storyboard x:Key="VideoSB">
        <MediaTimeline Source="Media\bee.wmv"
Storyboard.TargetName="media"/>
    </Storyboard>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <MediaElement x:Name="media" Margin="5" LoadedBehavior="Manual"/>
        <StackPanel Orientation="Horizontal" Grid.Row="2"
Background="LightSteelBlue">
            <Button Margin="5" Width="50">Play</Button>
            <Button Margin="5" Width="50">Pause</Button>
            <Button Margin="5" Width="50">Stop</Button>
        </StackPanel>
    </Grid>
</Window>
```

Note que el elemento media se ha definido con LoadedBehavior con valor Manual por lo que el vídeo no va a reproducirse cuando comience la aplicación. Para echarlo a andar es necesario iniciar la animación definida por el StoryBoard guardado en los recursos con llave VideoSB. Observe que en el Listado 16-6 hemos suprimido todos los métodos que habíamos anotado a los eventos de clic en los botones.

En el Listado 16-7 hemos incluido desencadenadores EventTriggers que aplican cada una de las operaciones BeginStoryBoard, PauseStoryBoard y StopStoryBoard cuando ocurra respectivamente un evento de clic en cada uno de los botones de nombre PlayButton, PauseButton y StopButton.

Listado 16-7 Inclusión de triggers que controlan al StoryBoard

```
<Window ...>
<Window.Resources>
    <Storyboard x:Key="VideoSB">
        <MediaTimeline Source="Media\bee.wmv"
Storyboard.TargetName="media"/>
    </Storyboard>
</Window.Resources>
<Window.Triggers>
    <EventTrigger RoutedEvent="Button.Click" SourceName="PlayButton">
        <EventTrigger.Actions>
            <BeginStoryboard Name="BeginMedia" Storyboard="{StaticResource
VideoSB}" />
        </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="PauseButton">
        <EventTrigger.Actions>
            <PauseStoryboard BeginStoryboardName="BeginMedia"/>
        </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="StopButton">
        <EventTrigger.Actions>
            <StopStoryboard BeginStoryboardName="BeginMedia"/>
        </EventTrigger.Actions>
    </EventTrigger>
</Window.Triggers>
<Grid>
    <Grid.RowDefinitions>...</Grid.RowDefinitions>
    <MediaElement x:Name="media" Margin="5" LoadedBehavior="Play"/>
    <StackPanel ...>
        <Button ... Name="PlayButton">Play</Button>
        <Button ... Name="PauseButton">Pause</Button>
        <Button ... Name="StopButton">Stop</Button>
    </StackPanel>
</Grid>
</Window>
```

En lecciones anteriores se han puesto los triggers siempre en el propio elemento que los desencadena o como parte de un estilo que se le aplique, sin embargo en el Listado 16-7 los triggers se han puesto en los recursos de la ventana. ¿Por qué? La causa está en el modo de funcionamiento de PauseStoryBoard y StopStoryBoard. Estos dos elementos detienen al elemento de tipo BeginStoryBoard cuyo nombre se indique como valor de la propiedad

BeginStoryboardName. Observe en el Listado 16-7 que al elemento BeginStoryboard le hemos dado nombre "SBStarted" y que luego en cada uno de los elementos PauseStoryboard y StopStoryboard nos hemos referido a este valor haciendo (BeginStoryboardName = "SBStarted"). Pero para poder usar el nombre SBStarted que se le dio al BeginStoryboard es necesario que los elementos PauseStoryboard y StopStoryboard estén en una misma colección de triggers junto al BeginStoryboard. En este ejemplo todos los triggers se han puesto en la colección Windows.Triggers.

Si ahora ejecutáramos la aplicación del Listado 16-7 se notaría que hay algo que aún no es deseable: Si hiciéramos clic sobre el botón de *Pause* y luego sobre el botón de *Play* el video se detiene primero pero luego reinicia nuevamente en vez de continuar a partir del punto en que se detuvo. Observe nuevamente el Listado 16-7 y note que al hacer clic en *Play* estamos cada vez reiniciando una animación con BeginStoryboard. La forma de lograr que una animación detenida con PauseStoryboard continúe es utilizando ResumeStoryboard. ¿Si sustituymos a BeginStoryboard por ResumeStoryboard dónde ponemos al entonces BeginStoryboard que inicia al video por primera vez? En el Listado 16-8 hemos indicado que esto se haga cuando se cargue la ventana (ocurrió el evento Window.Loaded). Note que hay dos acciones consecutivas asociadas a este evento

```
<BeginStoryboard Name="SBStarted" Storyboard="{StaticResource VideoSB}"/>
```

```
<PauseStoryboard BeginStoryboardName="SBStarted"/>
```

por lo que la animación se echa a andar pero inmediatamente a continuación se le hace pausa lo que significa que no se apreciará ningún efecto de reproducción (sea de audio o sonido). Esto significa que cuando se da al botón **Play** aunque realmente lo que se hace es un ResumeStoryboard de la sensación de que se ha comenzado la reproducción porque a ésta se le había dado una pausa inmediatamente a continuación de que se echó a andar al cargar la ventana.

Listado 16-8 Modificación de los triggers para que el botón Play provoque una continuación de la reproducción tras una pausa

```
<Window ...>
...
<Window.Triggers>
  <EventTrigger RoutedEvent="Window.Loaded" SourceName="window">
    <EventTrigger.Actions>
      <BeginStoryboard Name="SBStarted" Storyboard="{StaticResource VideoSB}"/>
      <PauseStoryboard BeginStoryboardName="SBStarted"/>
    </EventTrigger.Actions>
  </EventTrigger>
</Window.Triggers>
```

```

<EventTrigger RoutedEvent="Button.Click" SourceName="PlayButton">
  <EventTrigger.Actions>
    <ResumeStoryboard BeginStoryboardName="SBStarted"/>
  </EventTrigger.Actions>
</EventTrigger>
<EventTrigger ... SourceName="PauseButton">...</EventTrigger>
<EventTrigger ... SourceName="StopButton">...</EventTrigger>
</Window.Triggers>
<Grid>
  ...
</Grid>
</Window>

```

Utilizando elementos de tipo MediaElement manipulados como animaciones le invitamos a que incorpore sonidos a las plantillas de estilo de sus controles.

Lección 17 Gráficos 3D

Aunque desde la aparición de Windows las interfaces gráficas de interacción han ido evolucionando en calidad hasta Windows XP éstas han estado sobre un modelo bidimensional (la clásica ventana plana). Hasta la fecha han sido las aplicaciones de juegos las que han aprovechado las posibilidades tridimensionales que nos brinda cada vez más el hardware con el desarrollo de las tarjetas gráficas.

Otra de las grandes novedades de WPF es su estrecho vínculo con DirectX, la principal biblioteca de funciones y tipos 3D desarrollada para Windows. WPF incorpora de manera elegante y transparente un conjunto de recursos que reducen considerablemente la distancia entre el entorno bidimensional y el tridimensional.

En esta lección veremos como enriquecer nuestras aplicaciones con elementos que utilizan las potencialidades de la modelación 3D para mejorar la apariencia y la interacción de los clásicos controles bidimensionales.

17.1 Combinando los dos mundos

A modo de ejemplo inicial vamos a tomar la conocida ventana para dar formato a una unidad de disco (Figura 17-1) y vamos a mejorarle su apariencia y funcionalidad usando capacidades 3D.



Figura 17-1 Ventana de formato de unidades

Usted sabe que cuando damos clic sobre el botón **Aceptar** se despliega una segunda ventana de confirmación para esta operación. Este es un patrón de interacción utilizado en operaciones riesgosas (como ésta que implica pérdida de información) que requieren de confirmación. Dicho de otra manera: usted necesita hacer dos clics para realmente iniciar la operación.

La Figura 17-2 nos propone otra forma de definir esta ventana. Observe que hemos fundido en una misma ventana el inicio y la confirmación de la operación de formato.

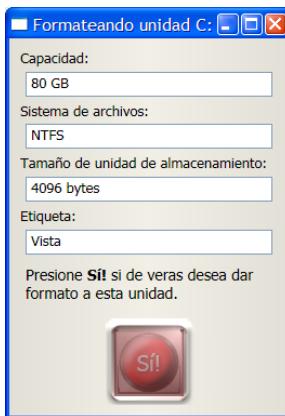


Figura 17-2 Ventana de la Figura 17-1 fundida con la ventana de confirmación.

¿Y qué pasó ahora con los dos Clics? Para acceder al botón de **Sí!** es necesario quitar la tapa (un primer clic) y luego dar clic sobre el botón (un segundo clic). Esta "tapa de seguridad" es similar a la que tienen algunos botones conmutadores de electricidad que evitan que uno los presione por accidente. Usaremos un efecto 3D para que al hacer clic sobre la tapa ella se abra y deje al descubierto el botón de **Sí!** como se muestra en la secuencia de la Figura 17-3.

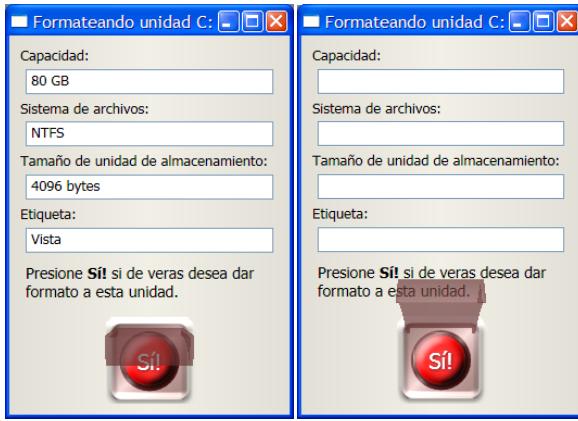


Figura 17-3 Dos momentos de la apertura de la Tapa de Seguridad tras el primer clic

Después de haber abierto la tapa de seguridad con el primer clic, el botón queda disponible para el **segundo clic** (Figura 17-4).

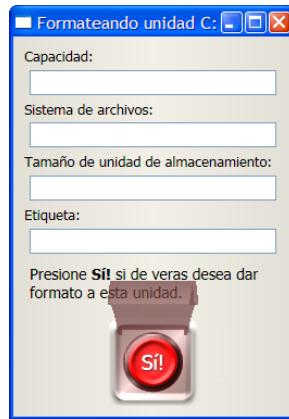


Figura 17-4 Botón presionado en el segundo clic

Ahora con WPF la variedad de posibilidades de interacción usando elementos tridimensionales, está limitada solamente por sus habilidades e imaginación. En las siguientes secciones mostraremos los elementos 3D que WPF nos brinda para desarrollar interfaces como la que hemos mostrado en este ejemplo.

17.2 Viewport3D y ModelVisual3D

En las lecciones anteriores hemos mostrado muchos de los elementos con los que usted puede conformar una interfaz de usuario. Entre ellos están los controles, las figuras, los paneles, los decoradores, etc. todos herederos de FrameworkElement, que es el tipo base de la mayoría de los elementos con los que usted puede conformar la apariencia de su aplicación. De igual modo usted puede incluir a los elementos 3D, utilizando el elemento especial Viewport3D heredero también de FrameworkElement, que es el encargado de procesar y proyectar los cuerpos tridimensionales en la superficie de las ventanas de su aplicación.

Los objetos tridimensionales que se ubican dentro del Viewport3D se agrupan en objetos de tipo ModelVisual3D. El Listado 17-1 muestra un primer Viewport3D donde hemos puesto un ModelVisual3D que describe la tapa de seguridad (Figura 17-5).

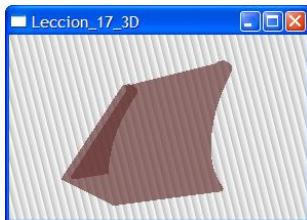


Figura 17-5 Tapa de seguridad

Locura total, dirá usted, si tenemos que escribir una extensa definición como la del ModelVisual3D del Listado 17-1. Esto se debe a que todos los cuerpos 3D se definen a partir de la unión de numerosos triángulos pequeños. Pero no se aflija que en esta lección le sugeriremos algunos atajos hacia formas más reducidas de especificar los cuerpos 3D. Por supuesto, que pronto comenzarán a aparecer herramientas visuales que lo ayudarán a hacer su diseño, y que generarán código en XAML que podrá usted integrar a sus aplicaciones.

17.3 Primitivas

El entorno 3D está compuesto en primer lugar por cuerpos tridimensionales que se definen en un GeometryModel3D. Estos objetos permiten indicar la forma del cuerpo mediante su propiedad Geometry de tipo Geometry3D. Al igual que las figuras bidimensionales que se pueden colorear con brochas y a las que se le puede transformar sus coordenadas de diferentes maneras, a los cuerpos 3D se le pueden indicar materiales mediante las propiedades Material y BackMaterial (de tipo Material) de GeometryModel3D y también se le pueden aplicar transformaciones mediante la propiedad Transform (que es de tipo Transform3D) de GeometryModel3D.

Además de los objetos tridimensionales la percepción del mundo que nos rodea depende de las luces que influyan sobre estos objetos y por supuesto, de nuestros ojos. En WPF, además de cuerpos, usted también puede poner diferentes tipos de luces (de tipo Light) dentro de un ModelVisual3D a las cuales también se les pueden aplicar transformaciones en el espacio tridimensional. La representación de nuestros ojos en este entorno es la cámara que se indica a través de la propiedad Camera de Viewport3D. Al final del Listado 17-1 usted podrá encontrar un ejemplo de cada uno de estos elementos.

Composición

Todos estos elementos siguen una lógica de composición basada en los tipos Viewport3D, Visual3D, ModelVisual3D, Model3D y Model3DGroup. La composición se aplica en el siguiente orden jerárquico:

- ② Todo Viewport3D está compuesto por una colección de elementos de tipo Visual3D que reúne capacidades de animación, de interacción con el ratón, y permite aplicar transformaciones de coordenadas a los cuerpos del entorno 3D. Esta colección de elementos se indica a través de la propiedad Children de Viewport3D. El tipo ModelVisual3D es heredero directo de Visual3D.
- ② Los objetos de tipo ModelVisual3D forman una estructura jerárquica de entornos tridimensionales. Cada uno con sus propios cuerpos y luces. Cada ModelVisual3D define completamente un entorno tridimensional que se indica en su propiedad Content de tipo Model3D. Los sub-entornos se asocian mediante la propiedad Children que es una lista de Visual3D. El término sub-entorno no refiere aquí a un concepto espacial, sino a una relación asociada con las operaciones que se aplican a estos entornos, como animaciones o transformaciones de coordenadas. Si por ejemplo usted transforma las coordenadas de un ModelVisual3D, la misma transformación se aplica recursivamente a todos sus ModelVisual3D hijos, en el orden jerárquico que usted haya definido.
- ② Los entornos tridimensionales finalmente están definidos por los objetos de tipo Model3D, que es el tipo base de las luces (Light), de los cuerpos de tipo GeometryModel3D y de los grupos de cuerpos: Model3DGroup. Estos últimos permiten agrupar cuerpos 3D como si fueran uno sólo.

En las siguientes secciones iremos describiendo con detenimiento a cada uno de estos tipos, a la vez que iremos conformando nuestro propio mundo tridimensional.

17.4 Mallas

Comencemos el estudio de cómo construir un mundo 3D con la descripción de las figuras tridimensionales que están compuestas básicamente por una gran malla de pequeños triángulos.

La primera de nuestras figuras tridimensionales será una de las más simples: el tetraedro de la Figura 17-6.

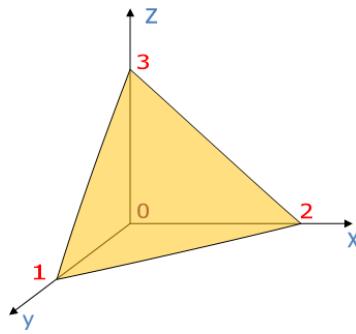


Figura 17-6 Tetraedro

Vértices y Caras

El tetraedro de la Figura 17-6 es una figura tridimensional formada por 4 puntos y 4 caras. Los puntos, expresados en coordenadas cartesianas (x,y,z) son:

Punto 0: (0,0,0)

Punto 1: (0,1,0)

Punto 2: (1,0,0)

Punto 3: (0,0,1)

Por su parte las caras están formadas por los triángulos planos que definen estos puntos:

Cara 0: 0 1 2

Cara 1: 0 2 3

Cara 2: 0 3 1

Cara 3: 1 3 2

Esta modelación 3D traducida a componentes WPF-3D es la que se muestra en el Listado 17-2. Observe que en la propiedad Positions se han indicado las coordenadas de los puntos y a su vez en TriangleIndices hemos puesto la definición de las caras usando estos puntos. La notación que se emplea en XAML para escribir en modo reducido los puntos y caras se basa en definir los elementos que conforman a un mismo punto o triángulo separados por un espacio, mientras que los propios puntos y triángulos se separan entre sí por dos espacios.

Listado 17-2 Tetraedro expresado en un MeshGeometry3D

```
<MeshGeometry3D Positions="0 0 0 0 1 0 1 0 0 0 1"
                  TriangleIndices="0 1 2 0 2 3 0 3 1 1 3 2"/>
```

Al definir cada triángulo usted debe además tomar en cuenta que cada triángulo en el mundo tridimensional tiene realmente dos caras. La forma de indicar cuál de las dos es la que queda hacia fuera es poniendo los puntos que definen al triángulo en el sentido de las manecillas del reloj respecto al punto de vista del que mira a la pantalla como si la estuviera viendo de frente en un eje de coordenadas como el de la Figura 17-6.

La Figura 17-7 muestra el tetraedro descrito en el Listado 17-2 .



Figura 17-7 Tetraedro en WPF

¿Cómo definir ahora un cubo o una esfera? Las caras del cubo son cuadradas, pero cada cuadrado es a su vez divisible en dos triángulos. Los cubos tienen 8 vértices y 6 caras cuadradas, como cada una está a su vez formada por 2 triángulos, la definición del valor de TriangleIndices tendría realmente 12 triángulos definidos en tríos de puntos.

¡Qué decir de la esfera! Usted siempre puede reunir un conjunto grande de pequeños triángulos que describan si no exactamente una esfera por lo menos una aproximación tan buena que la presencia de los triangulitos fuera imperceptible al ojo humano. ¿Pero entonces cuántos vértices y triángulos se necesitarían? El número es bastante largo y la definición casi tan extensa como la del Listado 17-1.

¿Y no hay otra manera de definir esto en XAML? No, lamentablemente la forma de definir un cuerpo tridimensional es construyéndolo con un conjunto numeroso de triángulos. Claro está, esto no está concebido así para que usted los ponga manualmente en XAML sino para que sea generado automáticamente por herramientas de diseño.

Software de Modelación

En general, XAML no está concebido para ser escrito completamente a mano. Microsoft además tiene en desarrollo un conjunto de herramientas que facilitan la generación de código XAML, entre ellas está el propio entorno de diseño de Visual Studio, el **Graphic Designer** que es una herramienta de dibujo y creación de brochas basada en DrawingBrush, el **Interactive Designer** que permite concebir casi en su totalidad la presentación de cualquier aplicación, incluyendo efectos visuales, animaciones, estilos, etc. Y aunque aún no se ha dado a conocer cuál de estas u otras herramientas darán soporte a la modelación 3D, es de suponer que la intención es que los entornos tridimensionales puedan ser completamente modelados en alguna herramienta especializada en ello.

Una buena noticia es que la mayoría de las aplicaciones profesionales de modelación ya existentes tratan las figuras tridimensionales basadas en los mismos conceptos primitivos (o semejantes) de vértices y caras. De alguna forma estas aplicaciones además exportan el resultado de la modelación 3D en diferentes formatos de archivos de texto a partir de los cuales usted puede tomar los puntos y caras de todos los cuerpos modelados.

No piense que el código de la tapa del botón de seguridad que se ha mostrado en el código del Listado 17-1 fue escrito a mano. La tapa la fue diseñada con una herramienta de diseño y luego su formato de representación lo hemos convertido a XAML. Es de esperar que en un futuro tales herramientas generen también XAML.

Otra de las opciones es aprovechar la flexibilidad de WPF en su jerarquía de tipos. ModelGeometry3D podría tener otros herederos. Las figuras elementales como cubos, esferas, conos, etc. y otras formadas por composición de estas pueden ser implementadas como tipos específicos de esta clase. Lamentablemente WPF aún no brinda tipos que den soporte a estas figuras básicas.

Para los ejemplos de esta lección hemos implementado cinco figuras elementales que generan automáticamente sus vértices y caras. La Figura 17-8 muestra estas cinco figuras. El código XAML se describe en el Listado 17-3.



Figura 17-8 Figuras generadas por código

Listado 17-3 Viewports construidos por Binding

```
<Window x:Class="WPF_3D.BasicGeometries"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ejemplo="clr-namespace:WPF_3D"
    Title="WPF_3D" Name="window" >
<Window.Resources>
    <ejemplo:PlaneBuilder x:Key="plane3D" Width="3" Height="3"/>
    <ejemplo:SphereBuilder x:Key="sphere3D" Radius="1" WidthSegments="100"
        HeightSegments="40"/>
    <ejemplo:ConeBuilder      x:Key="cone3D"      Radius1="1"      Radius2="0"
        WidthSegments="80"/>
    <ejemplo:CylinderBuilder x:Key="cylinder3D"   Radius1="1"      Radius2="1"
        WidthSegments="80"/>
    <ejemplo:WavePlaneBuilder x:Key="wave3D" WaveLength="1" Decay="0.05"
        Amplitude="0.5" Phase="0.0" Width="3.5" Height="4" />
```

```

        WidthSegments="140" HeightSegments="140" />
    </Window.Resources>
    <Grid Background="DarkBlue">
        <Grid.ColumnDefinitions>
            <ColumnDefinition/><ColumnDefinition/><ColumnDefinition/>
            <ColumnDefinition/><ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition/><RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Viewport3D DataContext="{StaticResource plane3D}">
            <Viewport3D.Camera>
                <PerspectiveCamera x:Name="camera"
                    Position="0,2,5" LookDirection="0,-2,-5" UpDirection="0,1,0"
                    FieldOfView="45" NearPlaneDistance="0.1"
                    FarPlaneDistance="100"/>
            </Viewport3D.Camera>
            <ModelVisual3D>
                <ModelVisual3D.Content>
                    <GeometryModel3D Geometry="{Binding Geometry}">
                        <GeometryModel3D.Material>
                            ...
                        </GeometryModel3D.Material>
                    </GeometryModel3D>
                </ModelVisual3D.Content>
            </ModelVisual3D>
        </Viewport3D>
        <Viewport3D DataContext="{StaticResource sphere3D}" Grid.Column="1">
            ...
        </Viewport3D>
        <Viewport3D DataContext="{StaticResource cone3D}" Grid.Column="2">
            ...
        </Viewport3D>
        <Viewport3D DataContext="{StaticResource cilinder3D}" Grid.Column="3">
            ...
        </Viewport3D>
        <Viewport3D DataContext="{StaticResource wave3D}" Grid.Column="4">
            ...
        </Viewport3D>
        <TextBlock ...> Plano </TextBlock>
        <TextBlock ...> Esfera </TextBlock>
        <TextBlock ...> Cono </TextBlock>
        <TextBlock ...> Cilindro </TextBlock>
        <TextBlock ...> Honda </TextBlock>
    </Grid>
</Window>

```

Observe que en el Listado 17-3 hemos puesto cinco objetos en los recursos de la ventana. Cada uno de estos objetos corresponde con una clase especializada en la construcción de mallas de triángulos para planos, esferas, conos y ondas tridimensionales. En el Listado 17-3 además usted puede notar que hemos puesto cinco Viewport3D con cada uno de estos objetos referidos en su DataContext, de modo que la propiedad Geometry del GeometryModel3D, que cada uno de estos Viewport3D contienen, está enlazada con una propiedad Geometry de estos objetos a través de la cual se obtiene la figura básica correspondiente.

17.5 Cámaras

Volvamos al ejemplo del tetraedro de la Figura 17-7. El Listado 17-2 donde se describe a esta figura no tiene todos los elementos necesarios para obtener la Figura 17-7. El principal elemento ausente es la cámara. El Viewport3D es como se dijo anteriormente una proyección del entorno 3D en un plano bidimensional que se dibuja en la ventana (ya que hasta ahora la tecnología de las pantallas de monitor de cualquier dispositivo es bidimensional). Dicho de otra manera el Viewport3D es, nada más y nada menos, que lo que está observando la cámara que pongamos en el mundo 3D (metáfora de nuestro ojo humano).

Las cámaras son elementos que tienen un conjunto de propiedades que permiten ubicarlas en el espacio tridimensional e indicar su ángulo y rangos de visión (Figura 17-9).

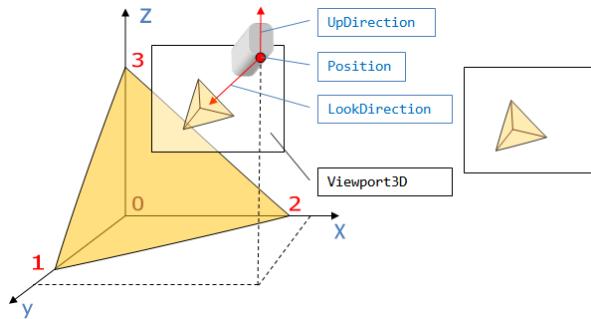


Figura 17-9 Cámara en un espacio tridimensional

Como muestra la Figura 17-9 las cámaras se definen por 3 propiedades principales:

Position: Indica la posición de la cámara en el espacio tridimensional.

LookDirection: Es un vector (x,y,z) que define la dirección hacia donde está orientada la cámara.

UpDirection: Vector que indica la inclinación vertical de la cámara. La Figura 17-10 muestra una cámara con un vector diferente. Observe cómo esta inclinación hace que la figura aparezca rotada en el Viewport3D.

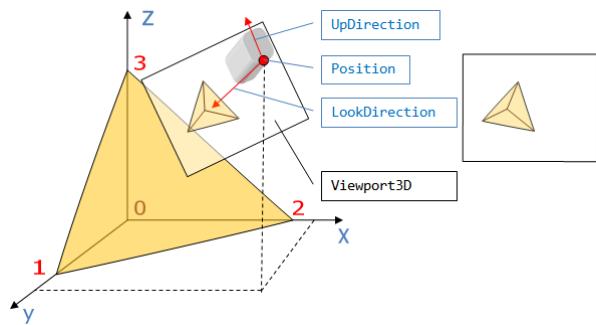


Figura 17-10 Cambio del Vector UpDirection de la cámara de la Figura 17-9

Adicionalmente, y por cuestiones de eficiencia, las cámaras incluyen otras dos propiedades: NearPlaneDistance y FarPlaneDistance. Ambas propiedades indican un rango de visión para la cámara. Los puntos que estén a una distancia de la cámara menor que el valor de NearPlaneDistance no serán procesados. De la misma manera, los puntos que estén más allá del FarPlaneDistance, tampoco serán procesados. Por ejemplo, suponga que su entorno 3D se desarrolla dentro de un apartamento donde la dimensión mayor de sus habitaciones no sobrepasa determinado valor X. Si se ubica la cámara en una de estas habitaciones, no es necesario procesar los cuerpos que estén en el resto de las habitaciones porque las paredes de todas maneras no dejarán verlos, así que usted puede dar a FarPlaneDistance el valor X y reducir así el tiempo de procesamiento de todo el entorno.

Tipos de cámara de proyección

Aunque el tipo base Camera en WPF permite que se puedan implementar diferentes tipos de cámara heredando de éste, actualmente se tienen sólo dos cámaras de proyección: La cámara ortográfica (OrthographicCamera) y la de perspectiva (PerspectiveCamera). La cámara ortográfica proyecta todo el entorno tridimensional en un plano que después se escala uniformemente al tamaño del Viewport3D.

La cámara de perspectiva por su parte proyecta el espacio tridimensional sobre un plano cuya dimensión horizontal la define un ángulo que se indica por la propiedad FieldOfView (campo visual). La Figura 17-11 muestra la diferencia entre ambas cámaras y la Figura 17-12 muestra el mismo ortoedro del Listado 17-2 usando ambas cámaras. El Listado 17-4 es el Listado 17-2 completado con una cámara de perspectiva.

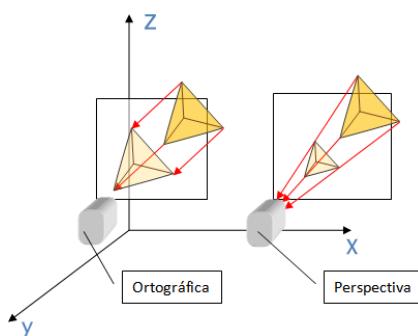


Figura 17-11 Modo en que los dos tipos de cámaras ortográfica proyectan sobre el Viewport3D

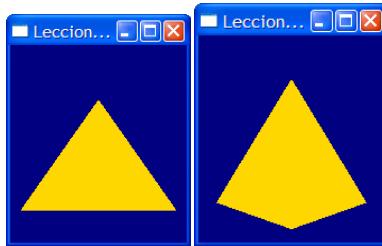


Figura 17-12 Ortoedro del Listado 17-2 visto con cámara ortográfica a la izquierda y de perspectiva a la derecha

Listado 17-4 Cámara de perspectiva

```
<Window x:Class="Leccion_17_3D.Window3"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Leccion_17_3D" Height="300" Width="300" >
<Grid Background="Navy">
<Viewport3D>
<Viewport3D.Camera>
<PerspectiveCamera
    NearPlaneDistance="1" FarPlaneDistance="3000"
    Position="-6 -6 3" UpDirection="0 0 1"
    FieldOfView="40" LookDirection="1 1 0"/>
</Viewport3D.Camera>
<ModelVisual3D>
<ModelVisual3D.Content>
<Model3DGroup>
<AmbientLight Color="White"/>
<GeometryModel3D>
<GeometryModel3D.Geometry>
<MeshGeometry3D
    Positions="0 0 0 1 0 1 0 0 0 1"
    TriangleIndices="0 1 2 0 2 3 0 3 1 1 3 2" />
</GeometryModel3D.Geometry>
<GeometryModel3D.Material>
<DiffuseMaterial Brush="Gold"/>
</GeometryModel3D.Material>
</GeometryModel3D>
</Model3DGroup>
</ModelVisual3D.Content>
<ModelVisual3D.Transform>
```

```

<ScaleTransform3D ScaleX="5" ScaleY="5" ScaleZ="5"/>
</ModelVisual3D.Transform>
</ModelVisual3D>
</Viewport3D>
</Grid>
</Window>

```

Hay que decir que lamentablemente WPF no incluye algo así como un Windows3D que en lugar de tener un Content de tipo FrameworkElement tenga un Content de tipo Visual3D. De este modo se podrían tener aplicaciones, donde todos los elementos fueran tridimensionales, en un Desktop de Windows también tridimensional.

17.6 Luces

Todo entorno 3D en WPF debe tener definida al menos una luz. Sin luz los cuerpos simplemente no se ven. Las luces no sólo hacen que los cuerpos se vean, las rutinas de procesamiento del mundo 3D calculan automáticamente las sombras de los cuerpos y los efectos de brillo. Hay cuatro tipos de luces. En la Figura 17-13 se muestran aplicadas las mismas al tetraedro. Estos tipos son **Ambiental**, **Direccional**, **Puntual** y de **Linterna**.

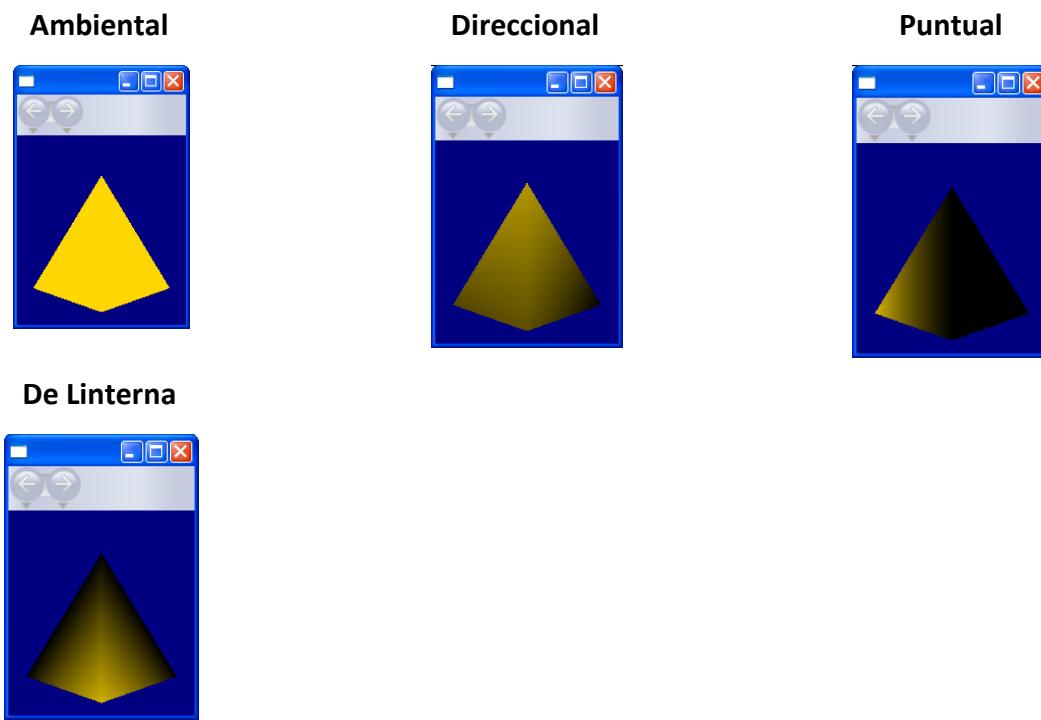


Figura 17-13 Tetraedro iluminado con las cuatro luces de WPF

Antes de pasar a describir estas cuatro luces debemos volver a las mallas de triángulos para comprender dos elementos importantes sobre la refracción de la luz en los cuerpos 3D:

- Para que un cuerpo se ilumine, la luz debe incidir en alguno de sus vértices. Para las luces ambiental o direccional esto no es inconveniente porque la luz siempre incidirá en todos los vértices de la figura, pero las luces: puntual y de linterna parten de un foco y se propagan en un rango de distancia determinado. Si en este rango la trayectoria de los haces de luz no tocan a un vértice, entonces WPF no lo identifica como que el cuerpo debe iluminarse y no produce el efecto de luz. Si quisieramos por ejemplo que iluminar directamente con una luz de linterna alguna de las caras del tetraedro, para que el efecto se perciba tendríamos que agregar nuevos puntos en el interior de dicha cara.
- La reflexión de la luz sobre un cuerpo está determinada por los *vectores normales* de los vértices. Cada vértice de una figura 3D tiene asociado un vector Normal que WPF calcula automáticamente a partir de las caras que lo tengan como vértice común. La Figura 17-14 muestra una cara y su vector Normal con una representación de cómo se refleja la luz al tocar el vértice, mientras que la Figura 17-15 muestra el vector Normal que WPF calcula a partir de dos caras que convergen en un mismo punto.

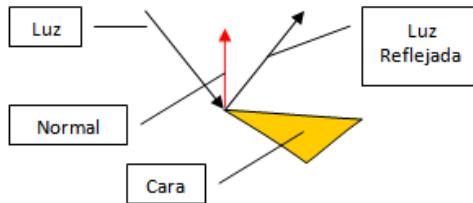


Figura 17-14 Dirección de reflejo de la luz respecto de la normal

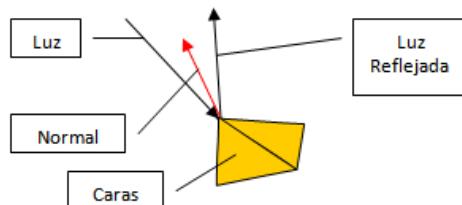


Figura 17-15 Normal calculada a partir de dos caras que convergen en un mismo punto

Aunque resulta muy cómodo que WPF nos ahorre el trabajo de calcular las normales. Esto no siempre es conveniente. Por ejemplo, seguramente que en los tetraedros de la Figura 17-13 usted no ha podido percibir bien las aristas del cuerpo. Esto ocurre porque la existencia de una única Normal por cada vértice hace que la luz se refleje de manera uniforme en todo el cuerpo dejando los bordes suavizados. Para poder ver con nitidez las aristas del tetraedro habría que repetir cada vértice tres veces, uno por cada cara que lo comparte, definir nuevamente los triángulos acorde con los nuevos puntos y luego definir una Normal diferente para cada uno de estos vértices utilizando la propiedad Normals de MeshGeometry3D como se muestra en el Listado 17-5. El resultado puede apreciarse en la Figura 17-16. Note cómo hemos vuelto a iluminar el tetraedro después de volver a definir sus puntos triángulos y normales. Observe que de todos modos con la luz ambiental

tampoco se observan las aristas con nitidez. Esto se debe a que esta luz no tiene ninguna dirección y por tanto las normales no se toman en cuenta.

Listado 17-5 Nuevo tetraedro con puntos repetidos y normales

```
<Page x:Class="Leccion_17_3D.Sample3D4"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Sample3D4" >
<Grid Background="Navy">
<Viewport3D>
<Viewport3D.Camera>
<PerspectiveCamera
    NearPlaneDistance="1" FarPlaneDistance="3000"
    Position="-6 -6 3" UpDirection="0 0 1"
    FieldOfView="40" LookDirection="1 1 0"/>
</Viewport3D.Camera>
<ModelVisual3D>
<ModelVisual3D.Content>
<Model3DGroup>
<SpotLight Color="White" Position="-3,-3,0" Direction="1,1,0"
    InnerConeAngle="3" OuterConeAngle="4" Range="20"/>
<GeometryModel3D>
<GeometryModel3D.Geometry>
<MeshGeometry3D
    Positions="0 0 0 0 0 0 0
              0 5 0 0 5 0
              5 0 0 5 0 0
              0 0 5 0 0 5"
    TriangleIndices="0 3 6 1 7 10 2 11 5 4 9 8"
    Normals="0 0 -1 0 -1 0 -1 0 0
              1 1 1 0 0 -1 0 -1 0
              -1 0 0 1 1 1 0 0 -1
              0 -1 0 -1 0 0 1 1 1"      />
</GeometryModel3D.Geometry>
<GeometryModel3D.Material>
<MaterialGroup>
<DiffuseMaterial Brush="Gold"/>
</MaterialGroup>
</GeometryModel3D.Material>
</GeometryModel3D>
</Model3DGroup>
```

```

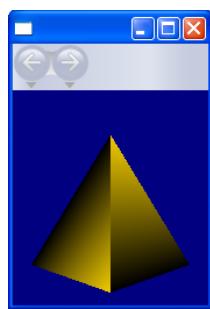
</ModelVisual3D.Content>
</ModelVisual3D>
</Viewport3D>
</Grid>
</Page>

```

Ambiental



Direccional



Puntual



De Linterna

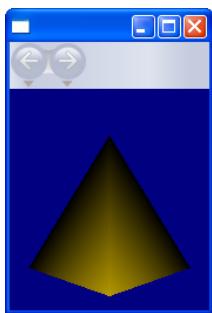


Figura 17-16 Luces sobre el tetraedro con normales cambiadas y vértices repetidos

En resumen los cuatro tipos de luces que hemos visto en las Figura 17-15 y la Figura 17-16 son los siguientes:

Luz ambiental (AmbientLight): Define una luz como la de una habitación y cuenta con una sola propiedad: Color.

Luz direccional (DirectionalLight): Es una luz de color Color que proviene de una dirección definida por la propiedad Direction como si proveniera de un punto muy lejano.

Luz puntual (PointLight): Es una luz que proviene de un punto que se indica por la propiedad Position y que se propaga con un rango (Range) de un color indicado por la propiedad Color, y que se atenúa con la distancia acorde con el valor de la propiedad LinearAttenuation.

Luz de linterna (SpotLight): Es un caso particular de lineal puntual que da el efecto del haz de luz de una linterna. Tiene las mismas propiedades de PointLight e incluye una dirección (Direction)

que define su trayectoria en un cono de luz que es más intenso en el ángulo indicado por la propiedad InnerConeAngle y menos intenso en el ángulo OuterConeAngle.

17.7 Materiales

Ya tenemos cámaras y luces pero para completar una buena definición de un entorno 3D no podían faltar los materiales. En la naturaleza los cuerpos están formados de disímiles tipos de materiales. Sin embargo, en cuanto al efecto que nos produce la luz sobre ellos, en WPF la mayoría de los materiales pueden agruparse en tres tipos de materiales que pueden ser combinados utilizando la propiedad Material para la parte anterior de las caras y BackMaterial para la parte posterior. Estos tipos de materiales son:

Difusor (DiffuseMaterial): Este material hace que cuando la luz incide sobre un vértice su intensidad se difunda y propague por las caras que parten de este vértice.

Reflejante (SpecularMaterial): Es una forma de representar los cuerpos brillantes. El material reflejante hace que la luz que incida sobre un vértice tome un brillo adicional en este punto que se corresponde con el ángulo de inclinación de la luz respecto de la normal en el punto. Es importante resaltar que este material debe ser combinado con el material difusor para que difunda también el brillo en las caras del cuerpo. La intensidad del brillo se indica en la propiedad SpecularPower. El efecto producido por este material se percibe mejor en figuras con muchos vértices.

Emisor (EmissiveMaterial): Este es un material peculiar que hace que los cuerpos emitan luz por sí mismos. Esta emisión ocurre siempre en todas direcciones pero esta es una luz que no ilumina a los demás cuerpos del entorno como las luces que describimos en la sección anterior.

Grupo de materiales (MaterialGroup): Adicionalmente usted puede emplear un grupo de materiales que combinen los anteriormente descritos. Recuerde que el material reflejante sólo tiene efecto en combinación con otros. La Figura 17-17 muestra tres esferas hechas con materiales diferentes. Observe cómo cada material modifica la forma en que la luz incide sobre los cuerpos.

Material difusor

**Grupo con material difusor y
reflejante**

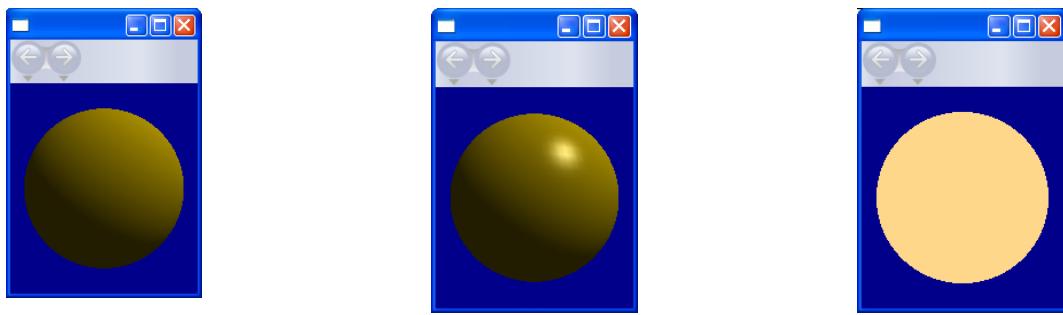


Figura 17-17 Diferentes tipos de materiales aplicados a una esfera

Note que la mayoría de los ejemplos son de un color dorado. Sin embargo los materiales tienen una propiedad Brush que permite llenar las caras de la figura utilizando cualquiera de las brochas descritas en la **Lección Brochas**.

Para llenar los cuerpos con brochas no sólidas, nuevamente debemos detenernos en las mallas de triángulos. Los materiales rellenan realmente a cada una de los triángulos que conforman el cuerpo, y la brocha que usted indique se empleará para llenar cada triángulo por separado. Si por ejemplo quisieramos pintar las caras laterales del tetraedro con una imagen como se muestra en la Figura 17-18 no bastaría con indicar un ImageBrush en la propiedad Brush, sino que además usted debe darle valor a la propiedad TextureCoordinates del MeshGeometry3D que define al tetraedro.



Figura 17-18 Caras del tetraedro llenadas con un ImageBrush

La propiedad TextureCoordinates de MeshGeometry3D es la que indica qué parte de la imagen representa cada vértice de la figura 3D. Por ejemplo, para el vértice del borde superior del tetraedro de la Figura 17-18 hemos indicado el valor 0,5,0 que corresponde con el punto medio-superior de la imagen. Cada par de valores en TextureCoordinates se corresponde con una posición en Positions. Para el vértice extremo izquierdo indicamos que debe colorearse la parte izquierda-inferior de la imagen: 0,1, para el vértice más cercano indicamos 0,5,0,5 correspondiente con la parte media-inferior de la imagen y finalmente para al vértice extremo derecho le hemos dado valor 1,1 que representa la parte derecha-inferior de la imagen (Listado 17-6).

Listado 17-6 Tetraedro lleno con una imagen
--

```

<Page x:Class="Leccion_17_3D.Sample3D6"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Sample3D6" >
<Grid Background="Navy">
    <Viewport3D>
        <Viewport3D.Camera>
            <PerspectiveCamera
                NearPlaneDistance="1" FarPlaneDistance="3000"
                Position="-6 -6 3" UpDirection="0 0 1"
                FieldOfView="40" LookDirection="1 1 0"/>
        </Viewport3D.Camera>
        <ModelVisual3D>
            <ModelVisual3D.Content>
                <Model3DGroup>
                    <SpotLight Position="-1,-1,0" Direction="1,1,0"
InnerConeAngle="1"
                        OuterConeAngle="2" Range="2"/>
                    <GeometryModel3D>
                        <GeometryModel3D.Geometry>
                            <MeshGeometry3D
                                Positions="0 0 0 0 1 0 1 0 0 0 1"
                                TriangleIndices="0 1 2 0 2 3 0 3 1 1 3 2"
                                TextureCoordinates="0.5 0 0.1 1 1 0.5 0.5"/>
                        </GeometryModel3D.Geometry>
                        <GeometryModel3D.Material>
                            <DiffuseMaterial>
                                <DiffuseMaterial.Brush>
                                    <ImageBrush ImageSource="Images/Windito.png"
Stretch="Fill"/>
                                </DiffuseMaterial.Brush>
                            </DiffuseMaterial>
                        </GeometryModel3D.Material>
                    </GeometryModel3D>
                    <Model3DGroup>
                </Model3DGroup>
            </ModelVisual3D.Content>
            <ModelVisual3D.Transform>
                <ScaleTransform3D ScaleX="5" ScaleY="5" ScaleZ="5"/>
            </ModelVisual3D.Transform>
        </ModelVisual3D>
    </Viewport3D>
</Grid></Page>

```

17.8 Transformaciones en 3D

En casi todos los listados de los ejemplos de esta lección usted puede encontrar el empleo de alguna transformación.

En el mundo tridimensional las transformaciones ayudan a simplificar el trabajo de calcular las coordenadas de los vértices. Es más simple poner todos los valores entre 0 y 1 y luego aplicar una transformación de escala ScaleTransform3D y acomodar las dimensiones de la figura al tamaño deseado empleando las propiedades ScaleX, ScaleY y ScaleZ.

La transformación de translación es también muy cómoda. Es mejor poner la figura que está definiendo en el centro de coordenadas (0,0,0) y luego trasladarla para la posición deseada utilizando las propiedades OffsetX, OffsetY y OffsetZ de TranslateTransform3D.

Las transformaciones de escala y translación en 3D son muy parecidas a las que ya se vieron en la [Lección Transformaciones](#), sin embargo la rotación en el mundo tridimensional se define de un modo distinto.

Las rotaciones se definen por el tipo RotateTransform3D, que a su vez tiene una propiedad Rotation donde se pueden indicar una de dos formas de rotación, una es empleando un QuaternionRotation3D que permite rotar la figura en las tres dimensiones a la vez y además se tiene el AxisRotation3D que permite rotar la figura en un ángulo determinado tomando un Vector (Axis) como eje de rotación alrededor del cual rota el cuerpo 3D (Figura 17-19).

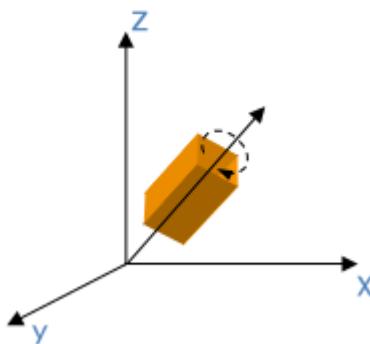


Figura 17-19 Prisma rotando alrededor de un vector

Al igual que en el mundo bidimensional se pueden hacer grupos de transformaciones 3D utilizando el tipo Transform3DGroup con lo cual se pueden aplicar distintas transformaciones a un mismo cuerpo.

Lo más significativo en las transformaciones 3D es su forma recursiva de propagarse en los ModelVisual3D. La estructura jerárquica con la que se definen los elementos 3D permite que al rotar, escalar o trasladar a un ModelVisual3D estas mismas transformaciones se apliquen a todos los cuerpos y luces que estén contenidos en éste y a los demás ModelVisual3D que de él desciendan.

17.9 Detección de Colisión (Hit Testing)

El mundo de modelos y objetos tridimensionales de WPF no solo trae consigo la capacidad de representar tales modelos en nuestras ventanas. La interacción del usuario con los objetos "por detrás" del ViewPort es posible gracias a lo que se da a conocer como detección de colisión o HitTesting.

Regresando al ejemplo del botón de seguridad de la Figura 17-3, vamos a lograr que el usuario pueda levantar la tapa de seguridad arrastrando el ratón sobre la tapa. El problema consiste en que la clase Visual3D brinda capacidades limitadas de interacción, puesto que se enfoca principalmente en el proceso de representar los modelos en el plano (rendering). Para los elementos del mundo 3D de WPF no se manejan conceptos tales como Click, MouseDown, KeyPressed, Focus, Layout, etc como sí se maneja en los controles planos, por tanto para lograr que un elemento 3D reaccione a interacciones como presionar un botón del ratón hay que implementarlo a nivel más básico capturando las interacciones y programando la funcionalidad de las mismas en el code-behind

La Figura 17-20 muestra un dialogo con el botón de seguridad que vimos en las primeras figuras de la lección. En el Listado 17-7 se describen algunos fragmentos del código XAML para lograr esta figura. Fíjese que los manejadores a los eventos MouseDown,MouseMove y MouseUp se definen en el Border que contiene al ViewPort.

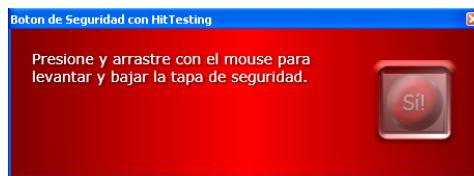


Figura 17-20 Botón de seguridad

Listado 17-7 Código XAML del botón de seguridad

```
<Window x:Class="WPF_3D.BotonHitTest" ...>
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition Width="130"/>
  </Grid.ColumnDefinitions>
  <Grid.Background>...</Grid.Background>
  <TextBlock ...>...</TextBlock>
  <Grid Grid.Column="1">
    <Border ...>...</Border>
```

```

<Border ...>...</Border>
<Button ...>Sí!</Button>
    <Border
        MouseDown="ViewPort_MouseDown"
        MouseMove="ViewPort_MouseMove"
        MouseUp="ViewPort_MouseUp"
        LostMouseCapture="ViewPort_LostMouseCapture">
        <Viewport3D Margin="0,3,0,0" Name="viewPort">
            <Viewport3D.Camera>
                <PerspectiveCamera
                    FarPlaneDistance="3000" LookDirection="-0,-,-1000"
                    UpDirection="0,1,0" NearPlaneDistance="1" Position="-130,-200,1000" FieldOfView="30"
                />
            </Viewport3D.Camera>
            <ModelVisual3D>
                <ModelVisual3D.Content>
                    <Model3DGroup>
                        <AmbientLight Color="White"/>
                        <DirectionalLight Color="#FFFFFF" Direction="-1,-1,-5" />
                        <PointLight Color="#FFFFFF" Position="-130,-100,1000" Range="3000" />
                        <Model3DGroup x:Name="tapa">
                            <Model3DGroup.Transform>
                                <Transform3DGroup>
                                    <RotateTransform3D>
                                        <RotateTransform3D.Rotation>
                                            <AxisAngleRotation3D x:Name="tapaRot" Axis="1,0,0" Angle="0"/>
                                        </RotateTransform3D.Rotation>
                                    </RotateTransform3D>
                                    <ScaleTransform3D ScaleY="0.6" ScaleZ="0.6" CenterY="-200">
                                        </ScaleTransform3D>
                                    </Transform3DGroup>
                                </Model3DGroup.Transform>
                                <GeometryModel3D x:Name="tapaMesh">
                                    <GeometryModel3D.Geometry>
                                        <MeshGeometry3D Positions="....." TriangleIndices=".....">
                                            </MeshGeometry3D>
                                    </GeometryModel3D.Geometry>
                                    <GeometryModel3D.Material>
                                        <MaterialGroup>
                                            <!--Brush="#6CDC"-->
                                            <DiffuseMaterial Brush="#A633"/>
                                            <SpecularMaterial Brush="#6FFF" SpecularPower="100"/>
                                        </MaterialGroup>

```

```

</GeometryModel3D.Material>
</GeometryModel3D>
</Model3DGroup>
</Model3DGroup>
</ModelVisual3D.Content>
</ModelVisual3D>
</Viewport3D>
</Border>
</Grid>
</Grid>
</Window>

```

En el code-behind se añaden los métodos del Listado 17-8.

Listado 17-8 code-behind del Listado 17-7 para lograr el efecto de subir y bajar la tapa al presionar y arrastrar el ratón.

```

void ViewPort_MouseDown(object source, MouseEventArgs e) {
    if (isMouseDown) return;
    if (e.LeftButton != MouseButtonState.Pressed) return;
    StartMoving(e);
}

void ViewPort_MouseMove(object source, MouseEventArgs e) {
    if (!isMouseDown) return;
    ContinueMoving(e);
}

void ViewPort_MouseUp(object source, MouseEventArgs e) {
    if (!isMouseDown) return;
    if (e.LeftButton != MouseButtonState.Released) return;
    FinishMoving(e);
}

void ViewPort_LostMouseCapture(object source, MouseEventArgs e) {
    CancelMoving(e);
}

bool isMouseDown;
bool isOpened;

void StartMoving(MouseEventArgs e) {
    Point mousePos = e.GetPosition(viewPort);
    PointHitTestParameters htp = new PointHitTestParameters(mousePos);
    HitTestResultCallback hitDeleg = delegate(HitTestResult result) {
        RayHitTestResult res = result as RayHitTestResult;
        if (res != null && res.ModelHit == tapaMesh) {
            isMouseDown = true;
            ratón.Capture(e.Source as IInputElement);
        }
    };
    viewPort.HitTest(htp, hitDeleg);
}

```

```

        return HitTestResultBehavior.Stop;
    }
    return HitTestResultBehavior.Continue;
};

VisualTreeHelper.HitTest(viewPort, null, hitDeleg, htp);
}

void ContinueMoving(MouseEventArgs e) {
    tapaRot.Angle = CurrentAngle(e);
}

void FinishMoving(MouseEventArgs e) {
    double angle = CurrentAngle(e);
    isOpened = angle < -90;
    AnimateTo(angle < -90 ? -120 : 0);
    isMouseDown = false;
    if (Ratón.Captured != null)
        Ratón.Captured.ReleaseMouseCapture();
}

void CancelMoving(MouseEventArgs e) {
    AnimateTo(isOpened ? -120 : 0);
    isMouseDown = false;
    if (Ratón.Captured != null)
        Ratón.Captured.ReleaseMouseCapture();
}

void AnimateTo(double toAngle) {
    DoubleAnimation anim =
        new DoubleAnimation(toAngle,
new Duration(TimeSpan.FromSeconds(0.5)));
    anim.AccelerationRatio = 0.5;
    AnimationClock clock = anim.CreateClock();
    clock.Completed += delegate {
        clock.Controller.Remove();
        tapaRot.Angle = toAngle;
    };
    tapaRot.ApplyAnimationClock(AxisAngleRotation3D.AngleProperty, clock);
}

double CurrentAngle(MouseEventArgs e) {
    Point mousePos = e.GetPosition(viewPort);
    double dist = (mousePos.Y - 40) / 60;
    dist = Math.Min(1.0, Math.Max(-1.0, dist));
    double angle = -Math.Acos(dist) / Math.PI * 180;
    angle = Math.Min(0, Math.Max(-120, angle));
    return angle;
}

```

El método StartMoving se ejecuta cuando se presione el botón izquierdo del ratón dentro del borde que contiene al Viewport3D. Por tanto, para comenzar a mover la tapa hay que asegurarse que esta acción ocurrió en el área que ésta ocupa dentro del Viewport3D. Lo primero es obtener la posición del ratón con relación al Viewport3D, esto es lo que se averigua con el código

```
Point mousePos = e.GetPosition(viewPort);
```

Luego se deben preparar los parámetros del método HitTest. La línea

```
PointHitTestParameters htp = new PointHitTestParameters(mousePos);
```

define el tipo de detección (test) que se debe realizar. La clase abstracta HitTestParameters es la base de dos clases que permiten detectar colisiones en entornos planos y con diferentes parámetros. PointHitTestParameters especifica que se debe utilizar un punto sobre la superficie de un objeto Visual para detectar qué otro objeto Visual o Visual3D aparece directamente debajo del punto. GeometryHitTestParameters permite describir toda un área definida por una figura geométrica cerrada para detectar las colisiones de dicha área con los objetos Visual que aparecen debajo. Una tercera clase RayHitTestParameters, que no hereda de HitTestParameters sino de HitTestParameters3D, permite detectar colisiones exclusivamente en un mundo 3D descrito por modelos 3D. Esta está definida por un punto (Point3D Origin) desde el cual parte un "rayo de colisión" y una dirección (Vector3D Direction) hacia la cual se dirige el rayo.

Los métodos de detección de colisiones se definen en la clase VisualTreeHelper. La versión sobrecargada más simple del método de detección es

```
public static HitTestResult HitTest(Visual reference, Point point);
```

Este método es suficiente mientras nos movemos en el mundo plano de los controles. Su comportamiento es simplemente quedarse con el objeto visual que aparezca inmediatamente debajo del punto que recibe como segundo parámetro. Este comportamiento evita que la detección pase a buscar en los modelos del mundo 3D. En el ejemplo hemos usado una segunda versión del método

```
public static void HitTest(Visual reference,  
    HitTestFilterCallback filterCallback,  
    HitTestResultCallback resultCallback,  
    HitTestParameters hitTestParameters)
```

Este método es una versión más general que la primera que mencionamos Busca colisiones, mientras no se le indique lo contrario, a través de los parámetros filterCallback y resultCallback.

El delegate HitTestFilterCallback describe métodos que reciben como parámetro un DependencyObject que es el blanco potencial de la detección de colisión y retorna un valor del

enumerativo HitTestFilterBehavior (Continue para especificar que el objeto es aceptable como resultado y que se debe buscar también entre sus descendientes, Stop para describir que se debe detener la detección, ContinueSkipChildren para especificar que el objeto es un blanco potencial pero sus objetos descendientes se deben excluir de la búsqueda, entre otros valores). El valor null utilizado en nuestro ejemplo se toma como Continue para todos los objetos encontrados en la búsqueda.

El delegate HitTestResultCallback se utiliza para ser llamado cada vez que se detecte una colisión. Recibe como parámetro un HitTestResult que describe la colisión de un objeto con el punto, geometría o rayo de colisión. Luego de procesar la colisión se retorna un valor del enumerativo HitTestResultBehavior, con los posibles valores Stop y Continue. El primero para detener la búsqueda y el segundo para continuar hasta la próxima colisión. En nuestro ejemplo, declaramos la variable hitDeleg de tipo HitTestResultCallback y le asociamos un método anónimo para poder acceder al parámetro e.

```
HitTestResultCallback hitDeleg = delegate(HitTestResult result) {
    RayHitTestResult res = result as RayHitTestResult;
    if (res != null && res.ModelHit == tapaMesh) {
        isMouseDown = true;
        Ratón.Capture(e.Source as IInputElement);
        return HitTestResultBehavior.Stop;
    }
    return HitTestResultBehavior.Continue;
};
```

El tipo HitTestResult es abstracto y el tipo real del parámetro en ejecución dependerá del tipo de colisión. En nuestro caso hemos utilizado RayHitTestResult. Observe que aunque la búsqueda se comenzó con un PointHitTestParameters, al pasar por la frontera entre el mundo 3D y el 2D, representada por el Viewport3D,, la búsqueda se transforma en detección por rayo de colisión. La propiedad ModelHit del tipo RayHitTestResult es el Model3D que colisionó con el rayo. Si su valor coincide con tapaMesh (es el GeometryModel3D de la tapa en el Listado 17-7) entonces significa que la búsqueda debe terminar, no sin antes capturar el ratón para controlar todos los movimientos de éste. Si no se ha colisionado con la tapa o ha ocurrido otro tipo de colisión se debe continuar la búsqueda.

El resto del code-behind no está relacionado con la detección de colisiones, sino con el control de la posición del ratón durante el arrastre y la animación que cierra o abre completamente la tapa al soltarla a medio camino.

En la Figura 17-21 se muestra la tapa al ser arrastrada por el ratón.

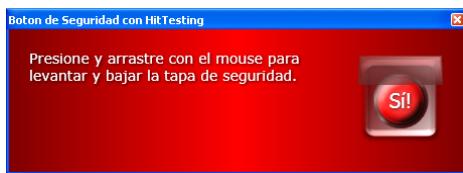


Figura 17-21 La tapa es arrastrada por el ratón, y su ángulo de apertura coincide con la posición aparente del cursor sobre ésta.

Lección 18 Documentos

En WPF se incluye el uso y tratamiento de documentos prácticamente al mismo nivel que los controles, figuras y demás elementos visuales. WPF ofrece diferentes capacidades de lectura de documentos, de almacenamiento de documentos, de control de la seguridad en el acceso al contenido, etc. Con estas capacidades WPF nos permitirá crear reportes en nuestras aplicaciones sin necesidad de conocer otra tecnología que no sea WPF, cuando antes era necesario generar archivos HTML o interactuar directamente con la interfaz COM del paquete Office, o en el mejor de los casos utilizar herramientas especializadas como CristalReport.

En esta lección haremos una incursión en este tema sin pretender abordar todas las capacidades que nos ofrece WPF.

En WPF hay dos formas de manipulación de documentos. Los documentos fluidos FlowDocument permiten describir el contenido de un documento en forma de secciones, párrafos, negritas e itálicas. Los documentos fijos o FixedDocument permiten describir documentos terminados, para los cuales se representan las páginas, las líneas las posiciones de cada carácter, trazos de líneas para dibujar tablas, etc.

18.1 Documentos Fluidos

Durante la fase de desarrollo de un documento, es muy probable que el autor no sepa del todo como quedará distribuido todo el contenido de cada página, párrafo y línea. En esta fase la modelación del documento esta compuesta por capítulos, epígrafes, tablas, párrafos, imágenes, estilos de texto, y texto propiamente dicho. La clase FlowDocument permite describir este tipo de documentos al estilo de lo que nos hace MS Office Word o HTML. Son los entornos donde se manipulan estos tipos de documentos los que generalmente permiten modificar la estructura visual del documento con solo reconfigurar el área de visualización.

Por ejemplo, un navegador de Internet trabaja con el formato HTML que permite describir documentos fluidos, orientados a una visualización variable. Si se cambia el tamaño del área de visualización del documento, se recomponen la apariencia del documento para que se adapte a las nuevas dimensiones, como muestran la Figura 18-1 y la Figura 18-2. Este es el tipo de documento útil para leer en la cada vez más versátil variedad de todo tipo de dispositivos, a la vez que permiten una representación para un formato fijo como impresoras y plotters.

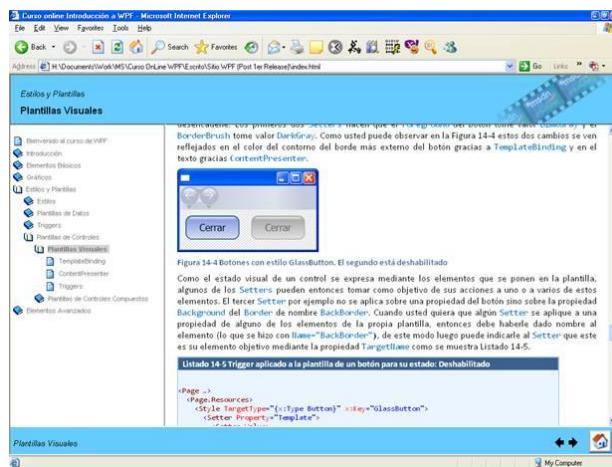


Figura 18-1 Visualización de un documento HTML en MS InternetExplorer con la ventana maximizada.

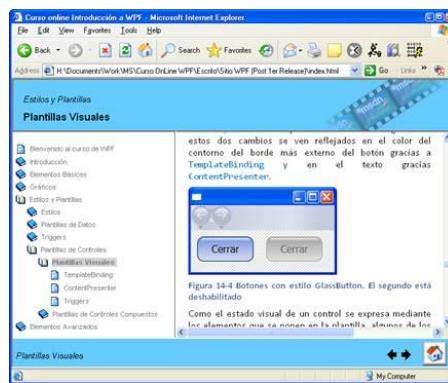


Figura 18-2 Visualización del mismo documento de la Figura 18-1 en una ventana más pequeña

Otro ejemplo lo tenemos con MS Office Word con la opción Configurar Página (**Page Setup**) que nos permite especificar las dimensiones de la página en la que finalmente estará representado el documento, como muestra la Figura 18-3. MS Office Word está concebido principalmente para crear documentos fijos. Es durante la edición de un documento que éste trata con documentos fluidos, a la vez que actualiza en tiempo real su representación como documento fijo, de manera que el usuario visualice por ejemplo cómo sería el resultado de imprimir el documento.

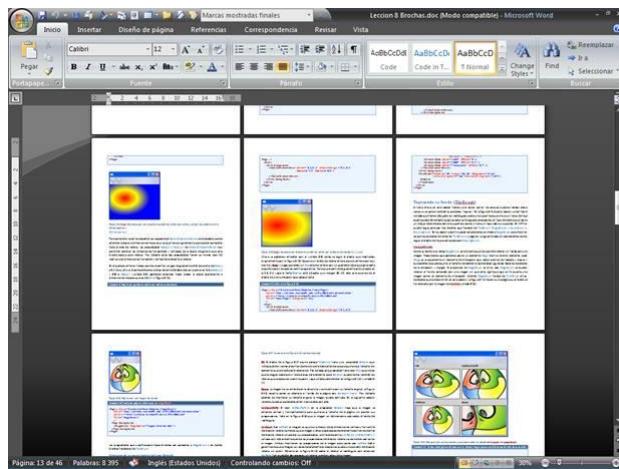


Figura 18-3 Visualización de un documento fluido con apariencia fija en MS Office Word 2007

En WPF se dispone de todos los elementos necesarios para representar documentos fluidos. Hay que destacar que las clases relacionadas con documentos fluidos no heredan de FrameworkElement, sino de FrameworkContentElement por lo que el uso simultáneo de documentos, controles y figuras debe seguir ciertas reglas que estaremos viendo durante la lección.

El flujo de texto en un documento está regido por ciertas reglas. Para obtener mayor rendimiento en la distribución de contenido de documentos, WPF en lugar de tomar la clase FrameworkElement como base, toma la clase FrameworkContentElement como clase base de la mayoría de las clases que veremos en esta lección. La distribución del contenido no lo realizan los paneles como ha ocurrido hasta ahora en el mundo de los controles sino una infraestructura de presentación de documentos.

Veamos el primer ejemplo de documento fluido en WPF. El Listado 18-1 muestra el código XAML de una aplicación que toma como página de inicio el documento fluido especificado en el Listado 18-2. La Figura 18-4 muestra el resultado de desplegar la aplicación.

Listado 18-1 Aplicación WPF que empieza con una ventana que muestra un documento fluido.

```
<Application x:Class="WPF_Documentos.App"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="PlantillasControlesFlow.xaml">
</Application>
```

Listado 18-2 Documento fluido descrito en XAML

```

<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
    <FlowDocument.Resources>
        <Style x:Key="Normal" TargetType="{x:Type Block}">
            <Setter Property="FontFamily" Value="Calibri, Arial"/>
            <Setter Property="FontSize" Value="12"/>
            <Setter Property="TextAlignment" Value="Justify"/>
        </Style>
        <Style x:Key="H2" TargetType="{x:Type Block}">
            <Setter Property="FontFamily" Value="Cambria, Times New Roman"/>
            <Setter Property="FontFamily" Value="Cambria, Times New Roman"/>
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="TextAlignment" Value="Left"/>
            <Setter Property="Foreground" Value="#385B86"/>
        </Style>
        <Style x:Key="PieFigura" TargetType="{x:Type Block}">
            <Setter Property="TextAlignment" Value="Left"/>
            <Setter Property="Foreground" Value="#385B86"/>
        </Style>
        <Style x:Key="CodeInText" TargetType="{x:Type Inline}">
            <Setter Property="FontFamily" Value="Consolas, Courier New, Courier"/>
            <Setter Property="FontSize" Value="11"/>
            <Setter Property="Foreground" Value="#385B86"/>
        </Style>
    </FlowDocument.Resources>
    <Section Style="{StaticResource Normal}">
        <Paragraph Style="{StaticResource H2}">
            Plantillas visuales
        </Paragraph>
        <Paragraph>
            En Windows.Forms Ud. puede cambiar el color de un control del mismo modo que
            podría cambiar el color de su coche: pintándolo de otro color. En WPF usted puede lograr el efecto de cambiarle a su coche no solo el color sino la carrocería completa. Una plantilla visual para un control WPF viene a ser como
            la carrocería para un coche (<Hyperlink NavigateUri="#fig14_1">Figura 14-1</Hyperlink>). Sin ella el coche es en principio funcional: arranca, gira, acelera, corre, frena, etc. pero no es atractivo. En WPF usted podrá cambiarle la apariencia a los controles que desee sin necesidad de definir nuevos controles (algo así como cambiar la carrocería sin tener que comprarse otro coche). Hay que decir que lamentablemente que en la irracionalidad actual
        </Paragraph>
    </Section>

```

muchos cambian de coche solo por cambiar de carrocería aún cuando el motor y el resto del mecanismo sigan funcionando de maravilla.

</Paragraph>

<Paragraph x:Name="fig14_1">

<Image Source="Images/carrito.jpg"></Image>

</Paragraph>

<Paragraph Style="{StaticResource PieFigura}">

Figura 14-1 Control Coche

</Paragraph>

<Paragraph>

Los controles de WPF se componen de dos elementos principales: su lógica funcional y su plantilla visual. La parte funcional se ocupa de mantener el estado del control y la interactividad con la lógica de negocio que lo esté utilizando, mientras que la plantilla visual se ocupa de darle color y otros efectos que le den atractivo y facilidad de uso. Con esta nueva estructura en el esquema de definición de controles de WPF usted puede cambiar fácilmente la apariencia de cualquier control con sólo cambiar su plantilla.

</Paragraph>

<Paragraph>

Todo control WPF tiene por defecto una plantilla que lo representa visualmente. Un control sin plantilla simplemente no se ve.

</Paragraph>

<Paragraph>

Las plantillas en WPF son objetos de tipo ControlTemplate. Una plantilla está "ligada" con un control a través de la propiedad Template que tienen todos los objetos derivados de la clase Control. Usted puede modificar la propiedad Template de un control asignándole una plantilla que se define directamente como se ilustra en el Listado 14-1, sin embargo esto no es lo más aconsejable. Usualmente queremos reutilizar una plantilla para que varios controles comparten un mismo estilo visual, es decir que tengan un aspecto parecido. Recordemos que en la Lección <Bold>Estilos</Bold> comentamos que la manera por excelencia de compartir estilos visuales es mediante el empleo de estilos que se colocan en los

recursos de los elementos que componen la aplicación. El Listado 14-2 nos muestra un esquema de definición de plantillas que mantendremos en todos los ejemplos de plantillas de esta lección.

```
</Paragraph>  
</Section>  
</FlowDocument>
```



Figura 18-4 Presentación en pantalla de un documento fluido

En el epígrafe **Visualización de Documentos** se analizan brevemente las funcionalidades de los íconos de la barra de herramientas que puede observar debajo del contenido del documento.

La estructura de un documento fluido es similar a un documento HTML, pero sigue reglas mucho más estrictas en su descripción, a la vez que tiene un número mucho menor de posibles elementos.

Observe en el Listado 18-2 que la raíz del documento XAML lleva por nombre `FlowDocument`, en lugar de `Page` o `Window` que son los que hemos usado en otras lecciones. Esto indica que se está describiendo un documento fluido. Este elemento `FlowDocument` puede contener recursos, al igual que otros elementos de WPF. En el Listado 18-2 se ha utilizado el diccionario de recursos `Resources` para colocar dentro del propio documento los estilos de texto que se utilizarán en éste. Por ejemplo, se ha usado una llave "Normal" (para imitar al estilo por defecto de MS Word) para asociar a algunos elementos de tipo `Block`, mientras que el estilo "CodeInText" se asociará a elementos de tipo `Inline`.

Las clases `Block` e `Inline` son las bases para componer documentos. Un bloque de texto (`Block`) es aquél que ocupa todo el ancho de una columna de texto. A su vez un bloque de texto está compuesto por varios elementos `Inline` como por ejemplo las palabras individuales del texto.

18.1.1 Block

La distribución de bloques de texto ocurre como si fuese en un StackPanel con orientación vertical. Es decir, como si se ubicase todo el documento en una única página de una sola columna como en la Figura 18-5. La diferencia se aprecian al aumentar el ancho de la ventana de visualización, como se muestra en la Figura 18-6.

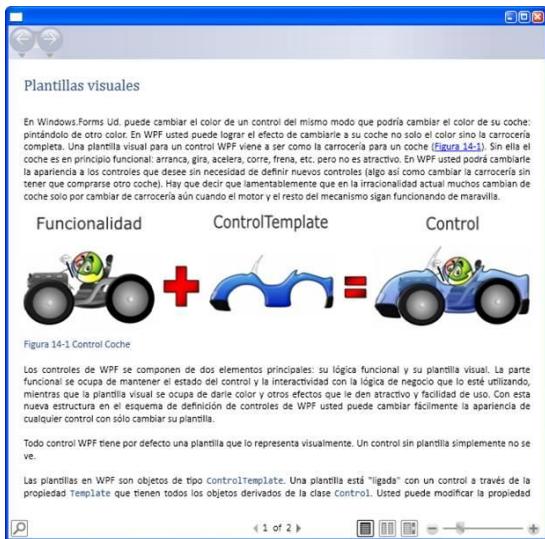


Figura 18-5 División y distribución de un documento en bloques

Al aparecer dos o más columnas la distribución es más compleja, parecida a un WrapPanel, pero con la salvedad de que en este caso un bloque de texto se puede dividir entre varias columnas o páginas, lo que no es posible en el mundo de los controles.

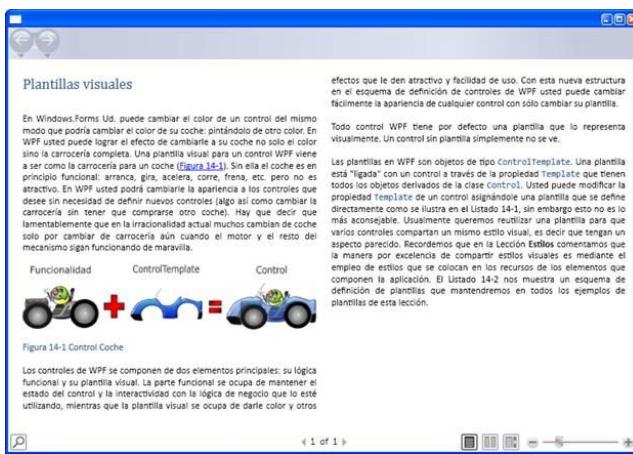


Figura 18-66 División y distribución de un documento en bloques

La clase **Block** es abstracta al igual que **Inline**, ambas heredan a su vez de la clase abstracta **TextElement**. Es en esta clase base de ambas donde están definidas las propiedades comunes a todo elemento de texto: **FontFamily**, **FontSize**, **FontStyle**, **FontWeight**, **Background**, **Foreground** (color del texto).

La clase Block define otras propiedades como BorderBrush y BorderThickness para definir marcos, Margin y Padding para separar el texto del borde y del contenido alrededor, BreakColumnBefore y BreakPageBefore para crear estilos de encabezamientos como títulos, que deben aparecer de primeros en la columna o página, FlowDirection para definir si la lectura ocurre de izquierda a derecha o a la inversa, LineHeight para la separación entre líneas, entre otras. La Figura 18-7 muestra algunas de las propiedades mencionadas anteriormente aplicadas al texto "Plantillas".

Figura 18-77 Algunas de las propiedades de Block aplicadas a un ejemplo

Los herederos de Block son Section, Paragraph, List, Table y BlockUIContainer. Cada uno representa un bloque de texto con sus características propias.

Paragraph

La clase Paragraph (utilizada en el Listado 18-2) es el más común de los bloques de texto, puede contener texto plano o formateado, así como también permite controlar cuándo el texto no debe ser dividido entre dos columnas o dos páginas (KeepTogether), cuándo se trata de un encabezamiento y el próximo párrafo debe mantenerse a continuación (KeepWithNext) la máxima cantidad de líneas huérfanas (MinOrphanLines), la sangría de la primera línea (TextIndent), etc. La propiedad Inlines de esta clase es una colección de instancias de Inline que describen el contenido del párrafo.

Section

La clase Section representa a las mayores estructuras de un documento, como los capítulos, epígrafes, etc. Su contenido nunca se compone directamente de elementos Inline, sino a su vez de otros elementos de bloque. Como caso particular una sección puede contener a otras secciones para darle estructura al documento. El principal aporte de Section a su clase base es la propiedad Blocks de tipo colección de Block.

Si se quiere poner un título a una sección, se debe poner el texto en el primer párrafo de la sección y mediante estilos hacer que el párrafo parezca visualmente como un título. No existe en WPF algo como <H2>...</H2>, lo que sería equivalente al primer párrafo del Listado 18-2

```
<Paragraph Style="{StaticResource H2}">
    Plantillas visuales
</Paragraph>
```

al cual le asociamos el estilo H2 de los recursos.

List

La clase List permite describir listas de elementos. Esta clase incluye las propiedades MarkerStyle para definir como lucen las viñetas o la numeración en la lista (los valores posibles pertenecen al enumerativo TextMarkerStyle entre los que se encuentran Disc, Circle, Box,

Decimal, UpperRoman, etc.). La propiedad MarkerOffset de tipo double define la distancia hacia la izquierda a la que se encuentran las marcas con relación al texto. StartIndex de tipo int permite indicar desde qué número o letra comienza la numeración. La propiedad ListItems es una colección de ListItem donde cada instancia determina un elemento de la lista. El Listado 18-3 muestra un documento con seis listas configuradas con diferentes estilos y distancias de los marcadores, como se muestra en el resultado de la Figura 18-8.

Listado 18-3 Documento con listas de elementos para utilizar viñetas y numeración

```
<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            FontFamily="Georgia, Bookman Old Style, Times New Roman"
FontSize="14"
    ColumnGap="1in" >
<FlowDocument.Resources>
    <Style x:Key="H2" TargetType="{x:Type Paragraph}">
        ...
    </Style>
</FlowDocument.Resources>
<Paragraph Style="{StaticResource H2}">Primera Lista</Paragraph>
<List>
    <ListItem>
        <Paragraph>
            Primer elemento de la lista
        </Paragraph>
        <Paragraph>
            Párrafo a continuación del primer elemento de la lista
        </Paragraph>
    </ListItem>
    <ListItem>
        <Paragraph>
            Segundo elemento de la lista
        </Paragraph>
    </ListItem>
    <ListItem>
        <Paragraph>
            Tercer elemento de la lista
        </Paragraph>
    </ListItem>
</List>
<Paragraph Style="{StaticResource H2}">Segunda Lista</Paragraph>
```

```

<List MarkerStyle="Circle">
...
</List>
<Paragraph Style="{StaticResource H2}">Tercera Lista</Paragraph>
<List MarkerStyle="Square">
...
</List>
<Paragraph Style="{StaticResource H2}">Cuarto Lista</Paragraph>
<List MarkerStyle="Decimal" MarkerOffset="12pt">
...
</List>
<Paragraph Style="{StaticResource H2}">Quinta Lista</Paragraph>
<List MarkerStyle="UpperRoman" MarkerOffset="1cm">
...
</List>
<Paragraph Style="{StaticResource H2}">Sexta Lista</Paragraph>
<List MarkerStyle="LowerLatin" MarkerOffset="0.5in">
...
</List>
</FlowDocument>

```

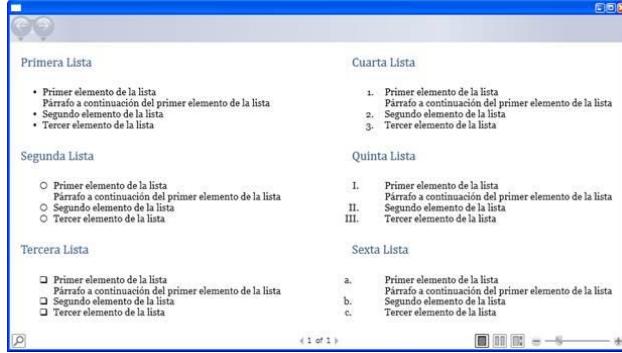


Figura 18-8 Presentación de un documento con listas de elementos

Table

La clase **Table** permite definir tablas en los documentos. Las tablas son los tipos de bloques de estructura más compleja. La estructura lógica de una tabla se muestra en el esquema de la Figura 18-9. La tabla contiene columnas (**TableColumn**) y grupos de filas (**TableRowGroup**). Las columnas no contienen ningún elemento, pero permiten especificar un ancho (**Width**) y el color del fondo (**Background**). Los grupos de filas contienen filas a su vez (**TableRow**) y estas contienen celdas (**TableCell**). Las tres clases **TableRowGroup**, **TableRow** y **TableCell** son todas herederas de **TextElement**, por lo que se les puede especificar un color de fondo, tipo de letra, tamaños estilos, etc. Dentro de una celda solo es posible ubicar un bloque de texto (**Block**).

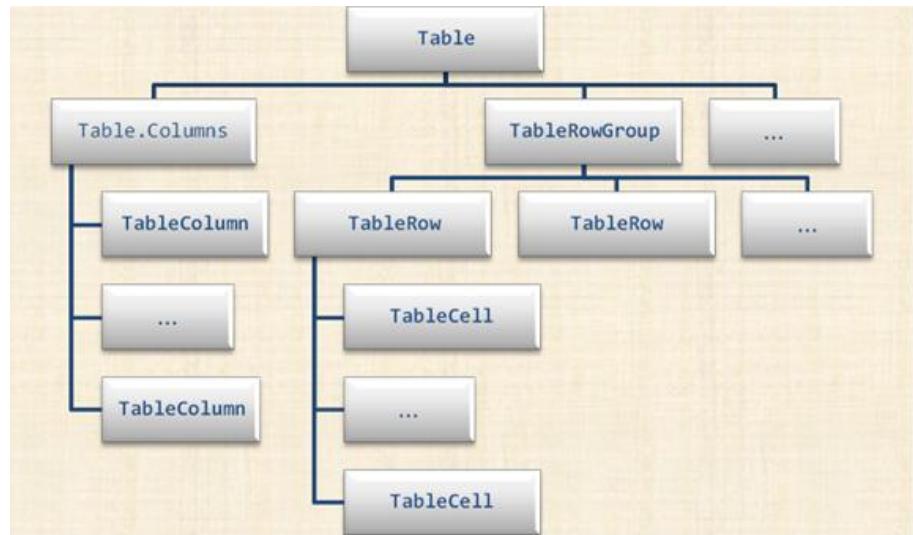


Figura 18-9 Esquema de Contenedor-Contenido de una tabla

El Listado 18-4 muestra un documento con una tabla que visualizaría un horario como el de la figura.

Listado 18-4 Descripción de un documento con una tabla en XAML

```

<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Table FontFamily="Calibri, Arial" FontSize="12" BorderBrush="CornflowerBlue" BorderThickness="1">
        <Table.Columns>
            <TableColumn Background="AliceBlue" Width="Auto"/>
            <TableColumn/>
            <TableColumn/>
            <TableColumn/>
            <TableColumn/>
            <TableColumn/>
            <TableColumn/>
            <TableColumn Background="#4820"/>
        </Table.Columns>
        <TableRowGroup Foreground="DarkBlue" FontSize="14">
            <TableRowGroup.Background>
                <LinearGradientBrush EndPoint="0,1">
                    <GradientStop Offset="0.0" Color="SteelBlue"/>
                    <GradientStop Offset="0.3" Color="AliceBlue"/>
                    <GradientStop Offset="1.0" Color="SteelBlue"/>
                </LinearGradientBrush>
            </TableRowGroup.Background>
        </TableRowGroup>
    </Table>

```

```

<TableRow>
    <TableCell></TableCell>
    <TableCell><Paragraph>Lunes</Paragraph></TableCell>
    <TableCell><Paragraph>Martes</Paragraph></TableCell>
    <TableCell><Paragraph>Miércoles</Paragraph></TableCell>
    <TableCell><Paragraph>Jueves</Paragraph></TableCell>
    <TableCell><Paragraph>Viernes</Paragraph></TableCell>
    <TableCell><Paragraph>Sábado</Paragraph></TableCell>
    <TableCell><Paragraph>Domingo</Paragraph></TableCell>
</TableRow>
</TableRowGroup>
<TableRowGroup>
    <TableRow>
        <TableCell><Paragraph>08:00am-10:00am</Paragraph></TableCell>
    </TableRow>
    <TableRow>
        <TableCell><Paragraph>10:00am-12:00am</Paragraph></TableCell>
    </TableRow>
    <TableRow>
        <TableCell><Paragraph>12:00am-02:00pm</Paragraph></TableCell>
        <TableCell ColumnSpan="6" Background="#4820">
            <Paragraph TextAlignment="Center" FontSize="14">Receso</Paragraph>
        </TableCell>
    </TableRow>
    <TableRow>
        <TableCell><Paragraph>02:00pm-04:00pm</Paragraph></TableCell>
    </TableRow>
    <TableRow>
        <TableCell><Paragraph>04:00pm-06:00pm</Paragraph></TableCell>
    </TableRow>
</TableRowGroup>
</Table>
</FlowDocument>

```

Solo se han rellenado algunas celdas para acortar el ejemplo.

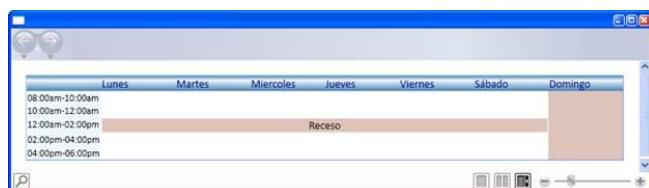


Figura 18-10 Visualización de un horario dentro de un documento

BlockUIElement

El elemento BlockUIElement es el que permite colocar controles, y otros elementos de interfaz de usuario, dentro de un documento. Imagínese si la documentación de las bibliotecas de clases de un determinado paquete de software incluyese además elementos interactivos que demuestren su funcionamiento. La calidad de asistencia que ofrecería una tal ayuda potenciaría la productividad en comparación a la que ofrecen las ayudas actuales.

El Listado 18-5 muestra un fragmento en XAML de un documento que contiene a su vez un control que se ha incrustado mediante un elemento BlockUIElement. Para simplificar se han dejado solo las partes de XAML necesarias para entender el ejemplo. En la Figura 18-11 se muestra el resultado de ejecutar este código, note cómo junto con el texto explicativo aparece en acción el control que muestra interactivamente lo que explica la documentación.

Listado 18-5 Código XAML de un documento fluido con controles incrustados

```
<FlowDocument  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">  
    ...  
    <Section FontFamily="Tahoma, Verdana, Arial" FontSize="12">  
        <Section Padding="4pt" Background="LightSteelBlue">  
            ...  
        </Section>  
        <Section>  
            ...  
        </Section>  
        <Section>  
            <Paragraph>Demostración</Paragraph>  
            <BlockUIContainer>  
                <Grid HorizontalAlignment="Left">  
                    <Grid.Resources>  
                        <Style TargetType="{x:Type TextBox}">  
                            <Setter Property="Margin" Value="2"/>  
                            <Setter Property="Padding" Value="1"/>  
                            <Setter Property="MinWidth" Value="160"/>  
                        </Style>  
                        <Style TargetType="{x:Type TextBlock}">  
                            <Setter Property="Margin" Value="0"/>  
                            <Setter Property="Padding" Value="0"/>  
                            <Setter Property="VerticalAlignment" Value="Center"/>  
                        </Style>  
                    </Grid.Resources>
```

```

<Border Background="LightSteelBlue" BorderBrush="SteelBlue"
    BorderThickness="1" CornerRadius="5">
    <Border.BitmapEffect>
        <DropShadowBitmapEffect/>
    </Border.BitmapEffect>
</Border>
<Border CornerRadius="5" Padding="12">
    <StackPanel>
        <Slider Name="slider"/>
        <DockPanel>
            <TextBlock DockPanel.Dock="Left">Value: </TextBlock>
            <TextBox Text="{Binding Value, ElementName=slider}" />
        </DockPanel>
        <DockPanel>
            <TextBlock DockPanel.Dock="Left">Minimum: </TextBlock>
            <TextBox Text="{Binding Minimum, ElementName=slider}" />
        </DockPanel>
        <DockPanel>
            <TextBlock DockPanel.Dock="Left">Maximum: </TextBlock>
            <TextBox Text="{Binding Maximum, ElementName=slider}" />
        </DockPanel>
    </StackPanel>
</Border>
</Grid>
</BlockUIContainer>
</Section>
</Section>
</FlowDocument>

```



Figura 18-11 Ejemplo de una documentación con controles incrustados.

18.1.2 Inline

Hemos visto hasta ahora los elementos que componen un documento por bloques. En esta sección veremos los elementos que componen al tipo de bloque párrafo. La clase `Inline` es la base de todo lo que pueda aparecer dentro de un párrafo. La propiedad `Inlines` de `Paragraph` es la que contiene una colección de instancias de tipo `Inline`. Esta clase hereda de `TextElement` y añade las propiedades `BaselineAlignment`, `FlowDirection` y `TextDecorations`. El Listado 18-6 usa la clase `Span` (que hereda de `Inline`) y nos muestra el uso de las propiedades `BaselineAlignment` y `TextDecorations` para lograr el resultado de la Figura 18-12.

```
Listado 18-6 Uso de BaselineAlignment y TextDecorations.

<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        FontSize="40" FontFamily="Cambria">
    <Paragraph FontStyle="Italic">
        <Span>x</Span><Span BaselineAlignment="Superscript">
FontSize="20">2</Span>
        <Span> + </Span>
        <Span>x</Span><Span BaselineAlignment="Subscript">
FontSize="20">2</Span>
    </Paragraph>
    <Paragraph>
        <Span TextDecorations="{x:Static TextDecorations.Baseline}">Baseline</Span>
        <Span TextDecorations="{x:Static TextDecorations.OverLine}">OverLine</Span>
        <Span TextDecorations="{x:Static TextDecorations.Strikethrough}">
            Strikethrough</Span>
        <Span TextDecorations="{x:Static TextDecorations.Underline}">Underline</Span>
    </Paragraph>
</FlowDocument>
```

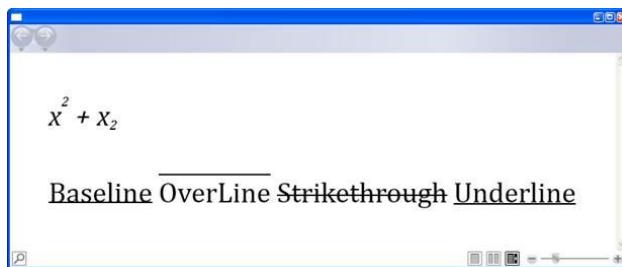


Figura 18-12 Uso de la línea base del texto para alinear y decorar el texto

Observe que la diferencia entre Baseline y Underline es mínima. En diferentes tipos de fuentes de texto esta diferencia se puede hacer más remarcada o más sutil. La más utilizada es Underline puesto que la línea aparece ligeramente por debajo de la parte inferior de las letras que no sobrepasan la línea base (Baseline) del texto.

Otros herederos de Inline son Run, Bold, Italic, Hyperlink, Underline, LineBreak, Figure, Floater y InlineUIContainer.

Run

La clase Run es el más simple de los contenidos textuales que se puede colocar en un documento. La propiedad Text le permite mostrar un texto plano. Sin embargo, aunque esta es la única propiedad que añade esta clase, hereda todos los beneficios de ser un TextElement, por lo que se pueden aplicar las propiedades FontFamily, FontSize, FontStyle, ... a un elemento de tipo Run. Este es el tipo que se utiliza cuando dentro de un párrafo se coloca un texto sin otro tipo de etiqueta. Por ejemplo <Paragraph>Hola</Paragraph> es equivalente a incluir el texto en una instancia de Run como en

```
<Paragraph><Run>Hola</Run></Paragraph>.
```

Span

Por otra parte, un Span permite agrupar varios elementos de tipo Inline, en su propiedad Inlines que es la única que añade a su clase base Inline. Por tanto además de poder aplicar al texto el estilo un estilo general, un Span puede contener otros elementos que pueden tener sus propios estilos.

Cuando se hace por ejemplo

```
<Span FontFamily="Arial"><Run FontWeight="Bold">Negrita</Run> Normal</Span>
```

se están incluyendo dos elementos Run a la propiedad Inlines del Span, uno indica aplicar Bold al texto Negrita y el otro pone el texto Normal sin negrita. Ambos se ponen en fuente Arial porque esto está especificado en el elemento Span que los contiene a ambos. De modo que esto se visualiza como a continuación

Negrita Normal

La clase Span es clase base de otras tres clases que nos dan efectos muy conocidos y utilizados Bold, Italic y Underline. Estas no añaden nada a la interfaz que heredan de Span, lo único que hacen es que sus constructores se configuran para forzar un determinad estilo dando el valor

deseado a la propiedad correspondiente de Span. Así por ejemplo<Bold> Texto en negrita </Bold>

no es mas que una forma abreviada de hacer

 Texto en negrita .

<Italic>...</Italic> es una forma abreviada de

...</Italic>.

y

<Underline>...</Underline> es una forma abreviada de

....

Hyperlink

Un Hyperlink es lo que conocemos en la Web como enlace, anchor o simplemente . Este elemento combina apariencia y funcionalidad y normalmente se utiliza para atraer la atención del usuario hacia una parte del texto desde la que se puede "navegar" hacia otro contenido u otro lugar en la Web. En WPF no se diferencia mucho de esto como muestra la Figura 18-13.



Figura 18-13 Estilo visual de los Hyperlinks

La propiedad NavigateUri, de tipo Uri es la que permite especificar la dirección hacia la que se debe navegar cuando el usuario active el Hyperlink. En XAML es fácil poner una dirección tanto local (nombre de otro archivo XAML que forme parte de nuestro proyecto y que quedará dentro del ensamblado) como una dirección Web. En el listado se muestran dos enlaces, uno navega al documento *PlantillaDatosFlow.xaml* descrito en el Listado 18-2 y el otro hacia el sitio de MSDN.

Listado 18-7 Enlaces en un documento

```
<FlowDocument  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
FontSize="14" FontFamily="Tahoma">
```

```

<Paragraph>
    Revise el documento
    <Hyperlink NavigateUri="PlantillasControlesFlow.xaml">Plantillas de Controles
    </Hyperlink> o el sitio de
    <Hyperlink NavigateUri="http://www.msdn.com">MSDN</Hyperlink>
</Paragraph>
</FlowDocument>

```

En la Figura 18-14 se muestra el resultado de la ejecución.



Figura 18-14 Dos hyperlinks para navegar a una dirección local y a una dirección en la web

Al activar el primer `Hyperlink`, la ventana realiza la navegación y muestra el contenido resultante como en la Figura 18-15 donde también se muestra cómo se puede regresar al documento anterior.

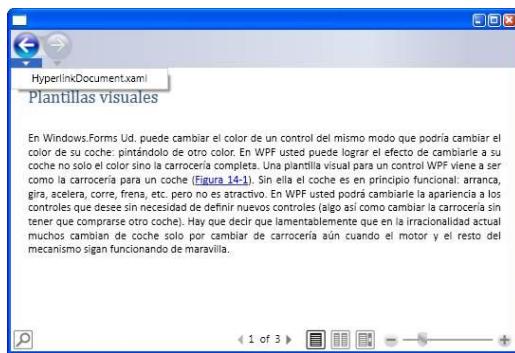


Figura 18-15 Luego de haber navegado, la ventana permite regresar al documento anterior, como en las aplicaciones Web.

A semejanza con los enlaces es la selección de donde se debe mostrar el contenido resultante de la navegación, con la propiedad `TargetName`. Por defecto es el navegador donde aparece el `Hyperlink` (ventana del browser o ventana de navegación de WPF) el que realiza la navegación, pero es posible escoger otra ventana como destino de la navegación o un marco (Frame). Un Frame es un elemento de WPF que permite incrustar un navegador dentro de una ventana.

En el Listado 18-8 se muestran tres Hyperlinks que navegan hacia el mismo documento, pero en tres navegadores diferentes. En la Figura 18-16 se muestra el resultado al activar los dos últimos Hyperlinks, ya que hemos visto que el resultado sería cambiar el contenido de la ventana por el nuevo documento. Al Frame le hemos puesto un color diferente para poder notar fácilmente el área que ocupa.

Listado 18-8 Selección del destino de la navegación

```

<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    FontSize="14" FontFamily="Tahoma">
    <Paragraph>
        <Hyperlink NavigateUri="PlantillasControlesFlow.xaml">
            Ver Pantillas de Controles aquí
        </Hyperlink>
    </Paragraph>
    <Paragraph>
        <Hyperlink NavigateUri="PlantillasControlesFlow.xaml" TargetName="frame">
            Ver Pantillas de Controles en el Frame
        </Hyperlink>
    </Paragraph>
    <Paragraph>
        <Hyperlink TargetName="otherWindow">
            Ver Pantillas de Controles en otra ventana
        </Hyperlink>
    </Paragraph>
    <BlockUIContainer>
        <Frame Name="frame" MinWidth="160" MinHeight="160"
Background="AliceBlue">
        </Frame>
    </BlockUIContainer>
</FlowDocument>

```



Figura 18-16 La ventana de la izquierda contiene los tres enlaces y un Frame de fondo azul claro. La ventana de la derecha tiene por nombre "otherWindow" para poderla seleccionar como destino de navegación.

Si no existe un navegador apropiado con el nombre especificado en TargetName, ocurre una excepción que detiene la ejecución.

Otra funcionalidad agregada a los Hyperlinks es la activación de comandos, a semejanza de cómo lo hacen los botones. Las propiedades Command, CommandTarget y CommandParameter.

LineBreak

El elemento LineBreak representa simplemente un cambio de línea dentro de un párrafo y no puede tener ningún contenido.

Floater y Figure

Las clases Floater y Figure heredan de la clase abstracta AnchoredBlock. Estas clases representan bloques de contenido que forman parte del flujo de texto o lo afectan de alguna manera. Por ejemplo, para obtener la Figura 18-17 se puede utilizar la clase Floater como en el Listado 18-9.

Listado 18-9 Uso de contenido flotante

```
<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
<FlowDocument.Resources>
    ...
</FlowDocument.Resources>
<Section Style="{StaticResource Normal}">
    <Paragraph Style="{StaticResource H2}">
        Plantillas visuales
    </Paragraph>
    <Paragraph>
        <Floater HorizontalAlignment="Right">
            <Paragraph x:Name="fig14_1" >
                <Image Source="Images/carrito.jpg" Width="300"
                    Stretch="Uniform"></Image>
            </Paragraph>
            <Paragraph Style="{StaticResource PieFigura}">
                Figura 14-1 Control Coche
            </Paragraph>
        </Floater>
    En Windows.Forms Ud. puede cambiar el color de un control del mismo
```

modo que

podría cambiar el color de su coche: pintándolo de otro color. En WPF usted puede lograr el efecto de cambiarle a su coche no solo el color sino la carrocería completa. Una plantilla visual ...

```
</Paragraph>
<Paragraph>
...
</Paragraph>
...
</Section>
</FlowDocument>
```

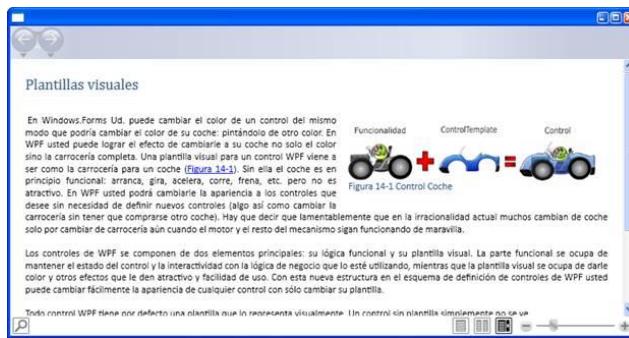


Figura 18-17 A la derecha del párrafo aparece un elemento flotante que contiene una imagen y un texto

La propiedad `HorizontalAlignment` permite especificar hacia cuál lado del bloque de texto se debe ubicar el contenido flotante.

La clase `Figure` es más potente pero más compleja de usar que `Floater`. Por ejemplo se puede lograr el efecto de la figura con `Figure` y no con `Floater`, puesto que el segundo solo puede moverse dentro del párrafo en el que está declarado (al ser un `Inline` solo puede aparecer dentro de un párrafo), mientras que con `Figure` es posible seleccionar con relación a qué se posiciona el contenido flotante. Para lograr la Figura 18-18 se substituyó el `Floater` del Listado 18-9 por `Figure` como en el Listado 18-10.

Listado 18-10 Uso de la clase `Figure`

```
...
<Paragraph>
<Figure HorizontalAnchor="PageCenter" VerticalAnchor="PageCenter">
<Paragraph x:Name="fig14_2" >
    <Image Source="Images/carrito.jpg" Width="300"
    Stretch="Uniform"></Image>
</Paragraph>
```

```

<Paragraph Style="{StaticResource PieFigura}">
    Figura 14-1 Control Coche
</Paragraph>
</Figure>
...
</Paragraph>
...

```



Figura 18-18 Uso de Figure para ubicar una imagen en el centro de la página

18.2 Documentos Fijos

Como hemos mencionado anteriormente los documentos fijos (FixedDocument) permiten fijar la estructura de un documento que ha sido desarrollado y revisado. Esta forma de modelar un documento es muy útil principalmente para publicar un contenido, como ocurre con los documentos en el formato PDF de Adobe, o cuando se imprime un documento a papel.

Para utilizar un contenido fijo se necesitan muchos menos recursos y potencia, puesto que cada detalle del contenido aparece ubicado exactamente donde debe, independientemente del resto del contenido del documento. Para manejar documentos fluidos es necesario un procesamiento más intenso para ubicar cada elemento de contenido en su lugar, en dependencia de las dimensiones que tenga el visualizador del documento (la ventana, marco, etc.). Aún con una estructura más inflexible, los documentos fijos permiten la búsqueda y selección de textos, una alta calidad de presentación y un nivel de portabilidad adecuado.

La clase FixedDocument es la raíz de todo documento fijo. Su único contenido se accede mediante la propiedad Pages que es una colección de PageContent. Esta clase a su vez maneja el contenido de la página dado por la propiedad Child de tipo FixedPage o por la propiedad Source de tipo Uri, en dependencia de donde se encuentre dicho contenido en relación al

documento. La clase `FixedPage` describe una página virtual, `Background` para definir el color de fondo, `BleedBox` y `ContentBox` describen las dimensiones del área útil de la página, y `Children` es una colección de `UIElement`, donde se describe el contenido de la página.

Generalmente el contenido de una página se describe dentro de un panel `Canvas` que permite posicionar los elementos en el plano.

En un próximo epígrafe (**Paquetes y XPS**) hablaremos de un formato de almacenamiento de documentos fijos que solo es cuestión de tiempo que se imponga en las plataformas Windows como formato para publicar e imprimir documentos.

18.3 Visualizando Documentos

Hasta ahora hemos hablado del modelo de descripción de documentos y hemos asumido en los ejemplos una apariencia por defecto a la hora de visualizar el contenido de los mismos. Todas las figuras de esta lección donde aparecen documentos fluidos tienen debajo una barra de herramientas para navegar por el contenido del documento como por ejemplo en la Figura 18-18.

Si ya estuvo jugando con esta barra, habrá notado que cada botón tiene una funcionalidad relacionada con la navegación a través del documento. En este epígrafe hablaremos de las diferentes formas de visualizar documentos en WPF.

18.3.1 Visualización de Documentos Fluidos

La forma más simple de visualizar un documento fluido es similar a la que nos presentan los navegadores de páginas web. Aquí se trata solo de distribuir verticalmente todo el documento, de forma continua, un párrafo detrás de otro, como en la Figura 18-19. El único control extra al documento es el `ScrollBar` a la derecha del contenido del mismo, que permite recorrer todo el contenido.



Figura 18-19 Presentación de un documento de manera continua

Para lograr esta figura se utilizó el código XAML del Listado 18-11.

Listado 18- 11 Presentación de un documento usando un FlowDocumentScrollView.

```
<Window x:Class="WPF_Documentos.VisualizandoDocumentos"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Visualizando Documentos">
<Grid>
    <FlowDocumentScrollView>
        <FlowDocument>
            ...
        </FlowDocument>
    </FlowDocumentScrollView>
</Grid>
</Window>
```

Esta forma de utilizar el FlowDocumentScrollView es poco realista porque requiere que el documento se especifique directamente en el mismo código XAML del visor. Lo más común será poner el visor en el código XAML como en el Listado 18- 12 pero cargar el documento en el code-behind para asociarlo al visor, como en el Listado 18- 13.

Listado 18- 12 Visor de documento sin contenido declarado en el código XAML.

```
<Window x:Class="WPF_Documentos.VisualizandoDocumentos"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Visualizando Documentos">
    <Grid>
        <FlowDocumentScrollView Name="viewer"/>
    </Grid>
</Window>
```

Listado 18- 13 Asociación de un documento al visor.

```
public VisualizandoDocumentos() {
    InitializeComponent();
    FlowDocument doc = new FlowDocument();
    Application.LoadComponent(
        doc,
        Uri("PlantillasControlesFlow.xaml",
        UriKind.RelativeOrAbsolute));
    viewer.Document = doc;
}
```

La propiedad Document utilizada en el código es de tipo FlowDocument.

La segunda forma de visualizar documentos fluidos es con el control FlowDocumentPageViewer, que es un visor por páginas de documentos fluidos. Si en el Listado 18- 12 cambia el visor FlowDocumentScrollView por FlowDocumentPageViewer, obtendrá el mismo resultado que la Figura 18- 20.



Figura 18- 20 Presentación de un documento por páginas.

Observe que este visor en lugar de un ScrollBar, presenta debajo una barra con dos controles. El primer control, al centro, substituye en funciones al ScrollBar, y permite navegar el documento por páginas. El segundo a la derecha representa el nivel de aumento del contenido (Zoom). Si producto del ancho del visor o del nivel de aumento se justifica poner el contenido de cada página en dos columnas, este control presenta el documento por columnas, como en la Figura 18- 21.



Figura 18- 21 Visualización de un documento en dos columnas.

Para este tipo de visor también existe la propiedad Document, pero en este caso es de tipo IDocumentPaginatorSource, y esta interfaz es implementada tanto por FlowDocument como por FixedDocument.* El tercer control para visualizar documento fluidos es FlowDocumentReader y es el que se utiliza por defecto cuando un navegador (NavigationWindow, o Frame) realiza una navegación hacia un documento, como en los primeros ejemplos de esta lección. En la Figura 18-22 se muestra el resultado de utilizar este tipo de visor en lugar de FlowDocumentScrollView



Figura 18- 22 Presentación en modo de Página de FlowDocumentReader

Con relación a FlowDocumentPageViewer, este visor añade dos funcionalidades nuevas: la búsqueda de texto y el cambio de modo.

La funcionalidad de búsqueda de texto se accede con la lupa a la izquierda de la barra de herramientas del visor. Al activar el botón de la lupa se muestra una pequeña barra de búsqueda como el de la Figura 18- 23. Al escribir un texto se puede buscar en el documento hacia delante y hacia atrás y queda señalado en el contenido la frase encontrada que corresponde con el texto buscado. También es posible configurar la búsqueda para tener en cuenta solo palabras completas, mayúsculas y minúsculas, entre otras opciones de localización.

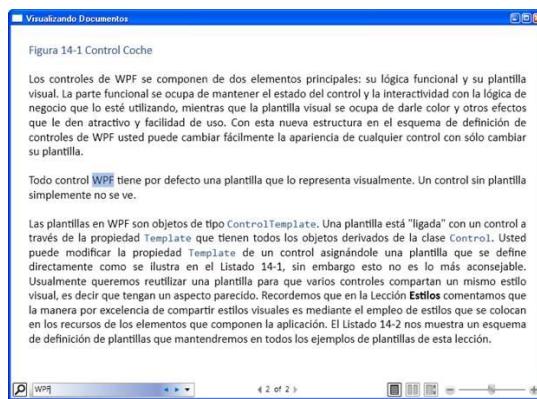


Figura 18- 23 Uso de la funcionalidad de búsqueda

La segunda funcionalidad que aporta este visor es el cambio en el modo de lectura. Por defecto se utiliza el modo de una página como en la Figura 18- 22. Pero a la izquierda del Slider de aumento se pueden apreciar tres íconos que representan los tres modos de visualización del visor, donde el primero es el descrito hasta ahora, similar al que nos ofrece FlowDocumentPageViewer. El segundo modo de dos páginas, permite visualizar el documento como en la Figura 18- 24. En este modo siempre se muestran dos páginas aunque todo el contenido quepa en una sola. El tercer modo es el modo Scroll, similar al que nos presenta FlowDocumentScrollView, como muestra la Figura 18- 25. Observe que aunque estos modos son similares a otros visores más simples, siempre se mantiene la funcionalidad de búsqueda y la posibilidad de cambiar de modo.

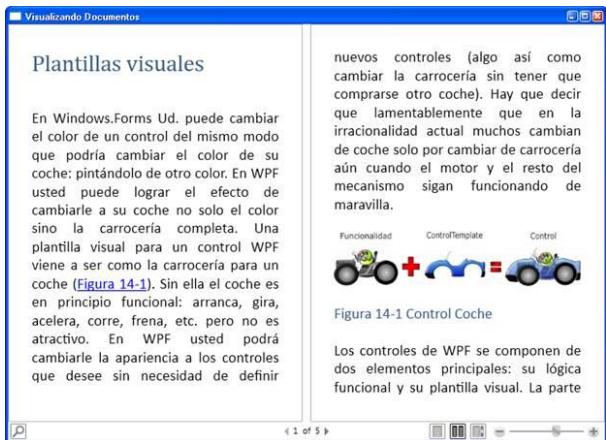


Figura 18- 24 Presentación en modo de dos páginas



Figura 18- 25 Presentación en modo Scroll

La propiedad Document de FlowDocumentReader es de tipo FlowDocument, y no puede mostrar documentos fijos.

La opción que debe utilizar en su aplicación dependerá de la funcionalidad requerida, pero siempre tenga en cuenta que más funcionalidad requiere de más rendimiento.

18.3.2 Visualización de Documentos Fijos

Ya vimos que el visor FlowDocumentPageViewer permite visualizar documentos fijos. Pero por la manera más común de manejar este tipo de documentos dejaremos para el epígrafe Paquetes y XPS el análisis de otro tipo de visor específico de documentos fijos.

18.4 Anotaciones

Otro de los recursos que mejoran considerablemente la experiencia de lectura son las anotaciones. Estamos acostumbrados a usar un marcador para resaltar las partes de un texto que más nos interesan o pegar notas con nuestras propias observaciones en nuestros libros. En esta sección veremos cómo hacer algo equivalente con WPF.

El primer paso será crear un documento (Listado 18- 14 y Figura 18- 26)

Listado 18- 14

```
<Window x:Class="WPF_Documentos.EjemploAnotaciones"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml>
<DockPanel LastChildFill="True">
    <FlowDocumentPageViewer>
        <FlowDocument >
            <Paragraph>
                La sombra es un efecto de luminiscencia que aparece sobre la
                superficie que no está directamente iluminada en los cuerpos. Puede
                simularse aplicando progresivamente un color oscuro
                semitransparente en el área donde se quiere crear el efecto de
                ausencia
                de luz. Esto hace que el área que dibujamos luzca voluminosa.
            </Paragraph>
        </FlowDocument>
    </FlowDocumentPageViewer>
</DockPanel>
</Window>
```

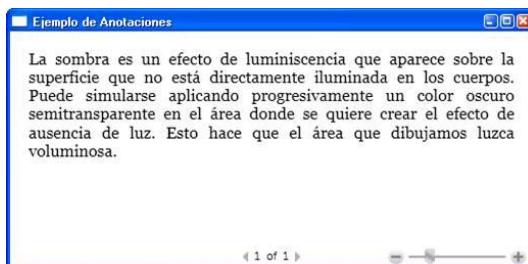


Figura 18- 26 Ejemplo de documento para colocar anotaciones

Las anotaciones de los documentos se aplican con comandos del tipo AnnotationServices del ensamblado PresentationFramework. AnnotationServices brinda comandos para crear tres tipos de anotaciones: enmarcado en amarillo (CreateHighlightCommand), pegatina de texto escrito a teclado (CreateTextStickyNoteCommand) y pegatina escrita a mano (CreateInkStickyNoteCommand). Análogamente tiene tres comandos que permiten eliminar las anotaciones del documento: ClearHighlightsCommand, DeleteStikyNotesCommand, y DeleteAnnotationsCommand. En el ejemplo del Listado 18-15 tenemos un menú que utiliza todos estos comandos (Figura 18- 27).

Listado 18- 15

```
<Window x:Class="WPF_Documentos.EjemploAnotaciones"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:ann="clr-
namespace:System.Windows.Annotations;assembly=PresentationFramework"
Title="Ejemplo de Anotaciones">
<DockPanel LastChildFill="True">
<Menu DockPanel.Dock="Top">
<MenuItem Header="Annotate">
<MenuItem Command="ann:AnnotationService.CreateHighlightCommand" />
<MenuItem Command="ann:AnnotationService.CreateTextStickyNoteCommand"/>
<MenuItem Command="ann:AnnotationService.CreateInkStickyNoteCommand"/>
<Separator/>
<MenuItem Command="ann:AnnotationService.ClearHighlightsCommand"/>
<MenuItem Command="ann:AnnotationService.DeleteStickyNotesCommand" />
<MenuItem Command="ann:AnnotationService.DeleteAnnotationsCommand" />
</MenuItem>
</Menu>
<FlowDocumentPageViewer Name="viewer">
<FlowDocument >
...
</FlowDocument>
</FlowDocumentPageViewer>
</DockPanel>
</Window>

```

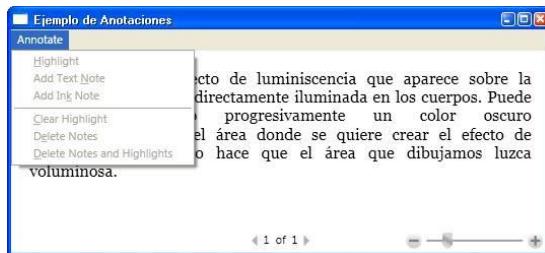


Figura 18- 27 Servicio de anotaciones deshabilitado

Sí, los comandos de menú aparecen deshabilitados. En el caso de las anotaciones los comandos se habilitan creando y habilitando un servicio de anotaciones. El chiste de las anotaciones es que estas aparezcan cada vez que se abra el documento pero que todas las marcas amarillas y pegatinas que pongamos no estén incluidas dentro del propio documento sino que se almacenen en otro archivo. Para esto es el servicio de anotaciones. Además del código XAML debemos agregar código C# que le indique a un servicio de anotaciones cuál es el visor de

anotaciones de donde obtendrá las marcas y pegatinas, luego hay que crear un archivo en el cual se almacenen las anotaciones que se adicionen (Listado 18- 16).

Listado 18- 16

```
private void OnLoad(object sender, EventArgs e) {
    service = new AnnotationService(viewer);
    file = new FileStream("Annotations.xml", FileMode.OpenOrCreate,
        FileAccess.ReadWrite);
    store = new XmlStreamStore(file);
    service.Enable(store);
}
private void OnClose(object sender, EventArgs e) {
    store.Flush();
    service.Disable();
    file.Close();
}
AnnotationService service;
FileStream file;
AnnotationStore store;
```

Note que el servicio de anotaciones se crea a partir de un visor de documentos y se habilita sobre un almacén de anotaciones (un archivo xml donde finalmente quedan guardadas las anotaciones). Anotando los métodos OnLoad y OnClose a los eventos Loaded y Closed (<Window ... Loaded="OnLoad" Closed="OnClose">...) el mecanismo de anotaciones sobre nuestro documento está completo (Figura 18- 28)

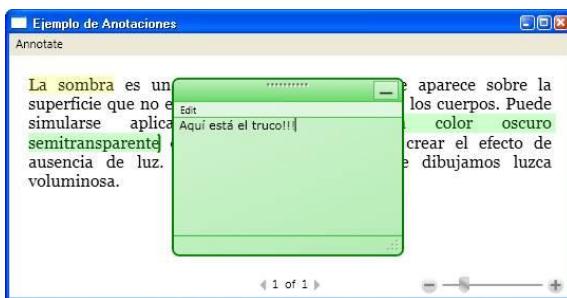


Figura 18- 28 Anotaciones de Enmarcado y Pegatina

Usted también puede cambiar la apariencia de estas pegatinas utilizando plantillas de estilo, brochas, transformaciones y efectos visuales como hemos hecho en el Listado 18- 17 lo que nos produciría una pegatina como la que se muestra en la Figura 18- 29.

Listado 18- 17

```
<Window x:Class="DocumentAnnotationWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ann= "clr-namespace:System.Windows.Annotations;assembly=
PresentationFramework" Title="Annotation sample"
    Loaded="OnLoad" Closed="OnClose">
<Window.Resources>
<Style TargetType="{x:Type StickyNoteControl}">
<Style.Triggers>
<Trigger Property="StickyNoteControl.StickyNoteType"
    Value="{x:Static StickyNoteType.Text}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate>
<Grid Name="TheGrid">
<Grid.RowDefinitions>
<RowDefinition Height="Auto" />
<RowDefinition />
</Grid.RowDefinitions>
<Border CornerRadius="4,4,0,0" BorderBrush="SteelBlue"
    BorderThickness="1,1,1,0">
<Border.Background>
<LinearGradientBrush EndPoint="0,1">
<GradientStop Color="#aFFF" Offset="0"/>
<GradientStop Color="#8FFD" Offset="0.2"/>
<GradientStop Color="#8FFD" Offset="0.9"/>
<GradientStop Color="#2000" Offset="1"/>
</LinearGradientBrush>
</Border.Background>
</Border>
<CheckBox HorizontalAlignment="Left" Margin="5"
Name="CB"
    Width="20" IsChecked="{Binding
        RelativeSource={RelativeSource TemplatedParent},
        Path=IsExpanded, Mode=TwoWay}" />
<RichTextBox Grid.Row="1" Name="PART_ContentControl"
    Background="LightYellow"/>
<Grid.BitmapEffect>
<DropShadowBitmapEffect Color="#8000" Direction="225"
    ShadowDepth="3" Softness="0.4"/>
</Grid.BitmapEffect>
```

```

<Grid.LayoutTransform>
  <RotateTransform Angle="-20"/>
</Grid.LayoutTransform>
</Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Trigger>
</Style.Triggers>
</Style>
</Window.Resources>
<DockPanel LastChildFill="True">
  <Menu DockPanel.Dock="Top">
    ...
  </Menu>
  <FlowDocumentPageViewer x:Name="viewer">
    <FlowDocument>
      ...
    </FlowDocument>
  </FlowDocumentPageViewer>
</DockPanel>
</Window>

```

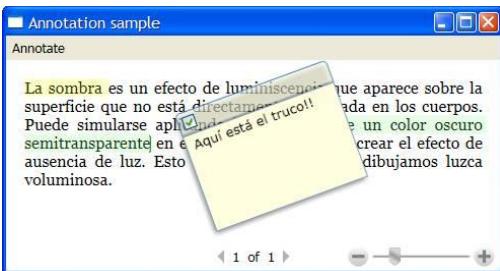


Figura 18- 29 Pegatina con estilo propio

18.5 Paquetes y XPS

El .NET Framework 3.0 junto con el nuevo sistema operativo Windows Vista propone varias tecnologías relacionadas con la publicación, impresión y manejo de documentos. Vamos a ver la tecnología de empaquetamiento y el formato de empaquetamiento de documentos XPS.

18.5.1 Paquetes

Se han desarrollado una gran variedad de formatos para almacenar información en archivos. Formatos para almacenar imágenes son los identificados con las extensiones bmp, gif, png y jpg.

Formatos para almacenar texto plano txt. Formatos para contenido Web como html, htm, aspx, js, css, asax, php, cgi etc. Formatos para documentos ricos como pdf, doc, ps, docx. Formatos para información compactada como zip, cab, rar, tar, gz, bz.

Algunos de estos formatos son propietarios y otros son estándares o recomendaciones de W3C, algunos usan codificación binaria y otros solo los caracteres ASCII para mantener compatibilidad en un entorno heterogéneo, algunos permiten almacenar un único tipo de contenido (imagen, texto, estructura) y otros permiten almacenar otros archivos cada cual con su formato específico.

Un paquete en WPF, tiene un doble propósito desde el punto de vista del almacenamiento de información en el sistema de archivos de Windows. El primer propósito es proveer un formato estándar de crear un único archivo estructurado y conformado de otros diferentes archivos, como ocurre cuando creamos un archivo comprimido de una carpeta que tiene documentos, imágenes, y a su vez otras carpetas anidadas. El segundo propósito es crear un tratamiento transparente a la compactación, encriptación y seguridad digital en el contenido de un archivo. Esto nos permite crear nuestros propios formatos de almacenamiento de información en archivo, siempre que a su vez esta información se pueda expresar en términos de otros formatos de archivo.

Por ejemplo, un documento que contenga texto, imágenes, estilos etc., se puede separar en el texto plano, en los estilos aplicados a diferentes partes del texto, en las imágenes usadas en el documento, en la estructura para indizar el contenido, etc. Cada una de estas partes se puede almacenar en un formato conocido y estándar y todo a su vez se puede compactar en un archivo zip. Lo que le faltaría a este archivo compactado para ser considerado un documento es que la aplicación encargada de abrirlo y trabajar con él, lo vea como tal, y no solo como una colección de archivos que no tienen relación alguna entre sí. Hace falta entonces que este archivo compactado describa cómo se relacionan sus partes (por ejemplo que indique dónde va cada imagen dentro del texto).

Este es el objetivo de la clase Package del espacio de nombres System.IO.Package del ensamblado WindowsBase. Esta clase abstracta expone un API de manipulación de paquetes. Un paquete va a estar formado por las partes que conforman el contenido y por relaciones entre las mismas.

De momento la única implementación de esta clase abstracta es ZipPackage, una implementación del API de empaquetamiento que además de ofrecer una manera de empaquetar varios archivos en uno solo, nos comprime el contenido de cada uno.

Una parte de un paquete (Part) es un archivo con un formato conocido como de los que mencionamos anteriormente (imagen, texto, un documento XML). Cada parte de un paquete

puede contener información o metainformación sobre otras partes del paquete (por ejemplo autor, título, fecha de creación, cantidad de revisiones realizadas).

Una relación no es más que una referencia que se toma desde el paquete o desde una parte del paquete hacia alguna parte del paquete o hacia alguna dirección externa como un sitio web o un recurso de la red. Además, cada relación tiene asociado un tipo identificado por un string y un identificador opcional de la instancia de la relación.

Desarrollemos como ejemplo un paquete que represente un álbum de fotos simple, de forma que cada foto (imagen) pueda tener asociado información extra como el autor de la foto, el tema, el título y la fecha en que se tomó. La Figura 18- 30 muestra un diagrama de cómo se estructura el paquete. Como partes del paquete se almacenarán las imágenes (denotadas como Foto1, ..., FotoN en la figura) y los datos asociados a estas (Datos1, ..., DatosN). Para organizar la información dentro del paquete, se creará el tipo de relación identificado por "uri:fotoDelAlbum" entre el paquete y las imágenes, denotada con las flechas azules en la figura. Para asociar los datos a las imágenes, se creará el tipo de relación "uri:datos" entre las imágenes y los archivos de texto que contienen los datos asociados a cada imagen.

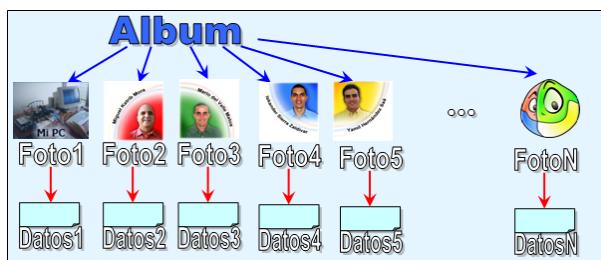


Figura 18- 30 Estructura interna de un paquete para almacenar un álbum de fotos.

Veamos las principales operaciones del API de paquetes de WPF. El código completo puede verse en los proyectos que acompañan a la lección.

Para el ejemplo se crearon dos clases: Album y Foto para representar al álbum y a las fotos. En el Listado 18- 18 se muestra como se crea un paquete.

Listado 18- 18 Creación de un paquete desde un fichero

```
public class Album {  
    public Album(string fileName) {  
        if (fileName == null) throw new  
            ArgumentNullException("fileName");  
        package = Package.Open(fileName, FileMode.OpenOrCreate,  
            FileAccess.ReadWrite);  
    }  
}
```

La clase Package tiene un método estático Open que utiliza un ZipPackage. A este método se le puede pasar como parámetro el nombre de un archivo. Al utilizar el modo OpenOrCreate se indica que si existe se abre este archivo y si no existe se crea uno por primera vez.

Una vez "abierto" un paquete es posible adicionarle partes. Esto es lo que hacemos con el código del Listado 18- 19.

Listado 18- 19 Al crear una parte se obtiene una entrada vacía en el paquete para comenzar a incluir información.

```
Uri uri = PackUriHelper.CreatePartUri(  
    new Uri(Guid.NewGuid().ToString(), UriKind.Relative));  
PackagePart part = album.package.CreatePart(uri,  
    MediaTypeNames.Image.Jpeg);  
album.package.CreateRelationship(uri, TargetMode.Internal,  
    "uri:fotoDelAlbum");...
```

Primero hay que escoger un Uri para introducir e identificar la parte dentro del paquete. El identificador puede ser estructurado como en el sistema de archivos. Por ejemplo, "/MiPC.jpg" o "/Imágenes/MiPC.jpg" son identificadores válidos. En el Listado 18-19 se genera un identificador con Guid.NewGuid() y se toma éste como string para darlo como identificador para cada nueva imagen.

El método NewGuid() de la clase Guid genera siempre un identificador diferente por cada llamada que se haga al mismo.

El método CreatePart de Package recibe como parámetros el identificador de la parte dentro del paquete (uri), el tipo mime del contenido de la parte (la clase MediaTypeNames contiene algunos predefinidos) y opcionalmente el modo de compresión utilizado para el contenido de la parte. El método de compresión por defecto es el valor NotCompressed (vea el enumerativo CompressionOption).

Para establecer una relación entre el paquete y una parte o un recurso externo, se utiliza el método CreateRelationship al que se le pasa como parámetros el identificador de la parte (uri), una indicación de si la relación es con una parte interna del paquete o con un recurso externo (TargetMode.Internal es lo que usamos en el Listado 18-19), y un identificador (string) del tipo de relación. En el Listado 18- 20 se muestra la creación de una relación entre dos partes del paquete.

Listado 18- 20 Creación de una relación entre ds partes

```
PackagePart imgPart = album.package.GetPart(uri);  
PackagePart datosPart = album.package.CreatePart(
```

```

    PackUriHelper.CreatePartUri(
        new Uri(Guid.NewGuid().ToString(),
        UriKind.RelativeOrAbsolute)),
    MediaTypeNames.Text.Plain, CompressionOption.Maximum);
imgPart.CreateRelationship(datosPart.Uri, TargetMode.Internal,
    "uri:datos", "datos");

```

Observe en el listado el método GetPart de la clase Package. Este método permite obtener la instancia de PackagePart identificada por el uri (de tipo Uri) en el parámetro. Observe también que se ha utilizado el modo de compresión máximo para la parte que almacena los datos de cada foto, ya que los contenidos de tipo texto admiten un radio de compresión mayor que los de tipo imagen en formato jpg. En la clase PackagePart también se encuentra un método CreateRelationship similar al de la clase Package, con el propósito de crear una relación de la parte imgPart para con la parte datosPart.

Para acceder a las relaciones del paquete o de una parte se pueden utilizar los métodos GetRelationship (para obtener una relación dado su identificador de instancia), GetRelationships (para obtener todas las relaciones del paquete o la parte) y GetRelationshipsByType (para obtener las relaciones de un tipo dado). Por ejemplo, en el Listado 18- 21 se itera sobre la colección de relaciones del tipo "uri:fotoDelAlbum" que fue el identificador utilizado en el Listado 18- 19 para crear las partes para las imágenes.

Listado 18- 21 Se pueden recorrer las relaciones de un mismo tipo del paquete o de una parte

```

fotos = new ObservableCollection<FotoAlbum>();

foreach (PackageRelationship rel in
    package.GetRelationshipsByType("uri:fotoDelAlbum"))
{
    fotos.Add(new FotoAlbum(this, rel.TargetUri));
}

```

En este caso, como la propiedad TargetMode tiene valor Internal, la propiedad TargetUri es a su vez un identificador de otra parte dentro del paquete. Esta otra parte la podemos obtener con el método GetPart.

En el Listado 18- 22 se utiliza el método GetRelationship para obtener una relación que tenga un identificador de instancia asociado. El método RelationshipExists permite conocer si existe en el paquete o en la parte una relación con el identificador dado, al igual que el método PartExists de Package permite conocer si existe una parte en el paquete dado su uri.

Listado 18- 22 Se puede obtener una relación si tiene un identificador de instancia asociado

```

PackagePart imgPart = ...;

if (imgPart.RelationshipExists("datos")) {
    PackageRelationship relaciónDatos = 
        imgPart.GetRelationship("datos");
}

```

Para poder leer el contenido de una parte, o para incluir información en una parte del paquete, se debe acceder al flujo ([Stream](#)) de ésta con el método [GetStream](#), como aparece en el Listado 18- 23.

Listado 18- 23 Uso del método GetStream para llegar al contenido de una parte

```

PackagePart part = ...;

Stream stream = part.GetStream();

```

Luego de desarrollar todo el API necesario para representar el Álbum se puede desarrollar entonces una capa de presentación para mostrar y editar el álbum en WPF, como en la Figura 18- 31.

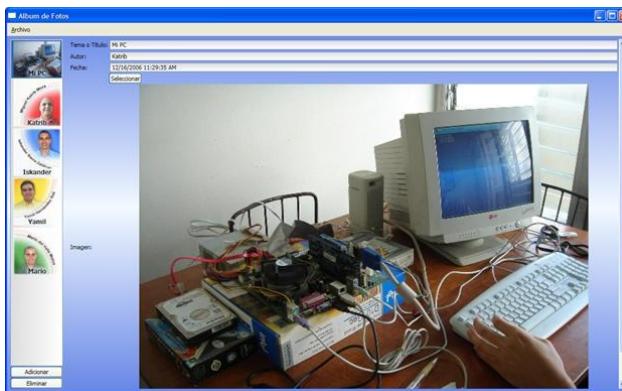


Figura 18- 31 Editando y revisando el editor del Álbum de Fotos en WPF.

Al crear el paquete a similitud del ejemplo anterior del Álbum de Fotos se puede salvar un archivo de cualquier extensión como por ejemplo MakingOfCursoWPF.album. Si desde el explorador de Windows se cambia la extensión a .zip, podemos ver que el archivo salvado como álbum es realmente un archivo compactado en formato zip, que dentro contiene un archivo por cada parte insertada en el álbum. En la Figura 18- 32. Este paquete contiene un archivo por cada parte (en el ejemplo son cinco imágenes y cinco archivos de texto para los datos de cada una), un archivo especial de nombre [ContentTypes].xml y una carpeta especial de nombre _rels. Los dos últimos mencionados forman parte del formato de ZipPackage.

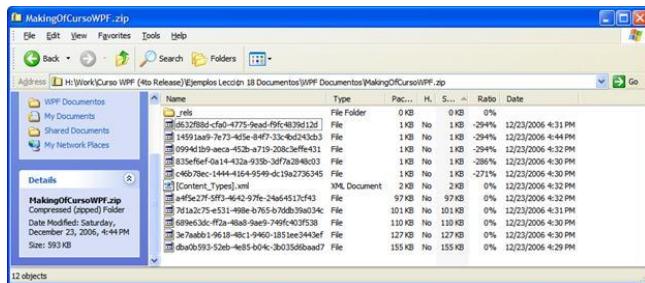


Figura 18- 32 Archivo ZIP que almacena el contenido del álbum de fotos

18.5.2 XPS

Si hemos incluido el epígrafe de Package en esta lección es porque el formato XPS, propuesto para la plataforma Windows Vista de almacenamiento de documentos fijos (y para impresión) utiliza este API de empaquetamiento para guardar las páginas del documento, las imágenes utilizadas en el documento, los datos del autor, fecha de creación y modificación, la imagen en miniatura (thumbnail) del documento, etc.

MS Office Word 2007 tiene una opción para salvar un documento con el formato XPS. La Figura 18- 33 muestra a Word 2007 antes de salvar una de las lecciones de este curso en formato XPS para ser utilizada en el ejemplo final del curso.

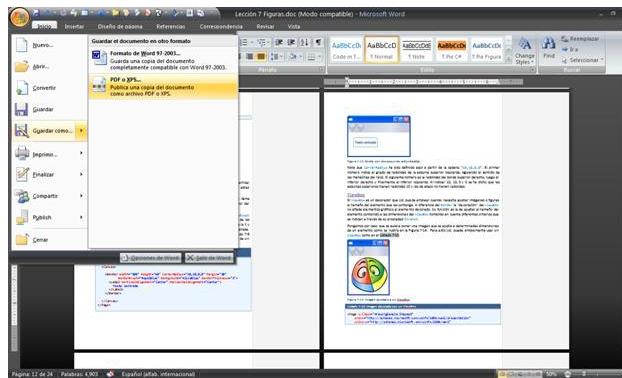


Figura 18- 33 Salvando un documento en MS Word 2007 en formato XPS

Al salvar el documento se obtiene un archivo con extensión xps. Realmente si cambiamos la extensión por zip, se puede abrir el archivo y apreciar la estructura del documento almacenado, como muestra la Figura 18- 34.

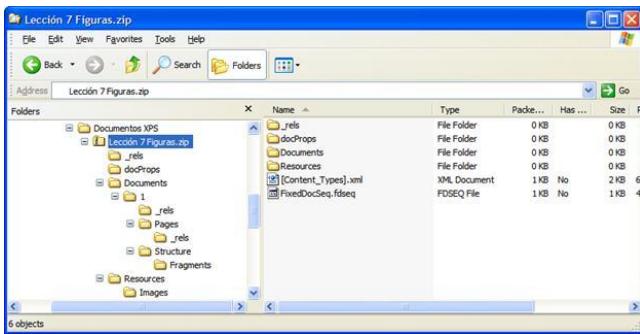


Figura 18- 34 Estructura interna de un documento XPS.

El paquete tiene tres partes fundamentales relacionadas directamente con el formato del documento. Una es la identificada por el uri "/docProps/core.xml" que es un archivo xml con los datos de autoría del documento como el autor, el título, y las fechas de creación y modificación. La segunda parte es una imagen identificada por "/docProps/thumbnil.jpeg" que representa una imagen en miniatura de la primera página del documento. La tercera parte es el documento propiamente dicho y su archivo raíz se puede observar en la figura bajo el nombre de "/FixedDocSeq.fdseq". Esta parte no es más que un documento xml que referencia los documentos (recuerde que en un XPS se puede almacenar varios documentos) dentro de la carpeta Documents que se ve en la figura. Para cada documento aparece una carpeta dentro de esta última, generalmente con una numeración secuencial, y dentro de esta aparece un índice, las páginas del documento (en un archivo separado por cada una) y la estructura lógica del documento (capítulos, secciones, epígrafes, ...).

Esta forma de almacenar documentos brinda por sobre todas las características una forma portable de publicar documentación de cualquier tipo. Los formatos de archivos utilizados son jpg, gif, png, bmp para las imágenes, odttf para fuentes embebidas en formatos OpenType y TrueTypeFont, xml para todo lo demás (texto, índices, etc.). Donde pudiera haber impedancia con otras plataformas es en el formato de las páginas, ya que se utiliza XAML. Sin embargo, solo se utiliza un conjunto muy reducido de clases y características de WPF para representar documentos.

Es de esperar que el formato XPS pase a ser el formato por excelencia de la plataforma Windows para la publicación e impresión de documentos. La instalación de WPF que se ha utilizado durante la preparación de este curso incluye un Document Writer para la impresión virtual de todo documento imprimible hacia XPS (Figura 18- 35).

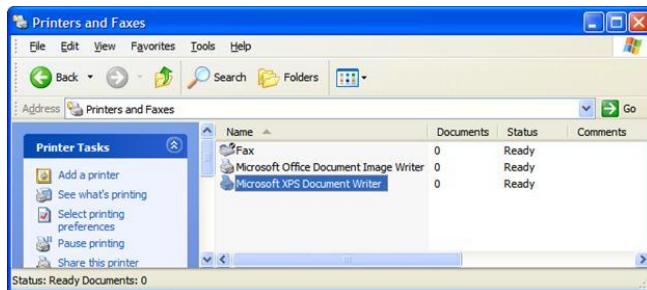


Figura 18- 35 En el Panel de Control se puede configurar el dispositivo de impresión para XPS

Visualizando el Documento

Es posible visualizar un documento XPS dentro del Explorador de Internet de Windows. Esta aplicación al intentar cargar el documento instancia al **XPS Viewer** que es una interfaz de usuario para realizar operaciones básicas con documentos XPS. La Figura 18-36 muestra el documento salvado en el epígrafe anterior visto con IE6.0.

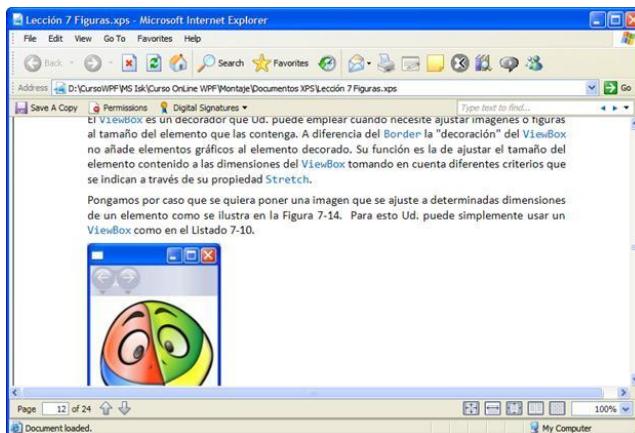


Figura 18- 36 Documento XPS dentro de Internet Explorer de Windows

Para visualizar un documento XPS desde nuestra aplicación WPF hay que utilizar el control **DocumentViewer** como en el Listado 18- 24.

Listado 18- 24 Uso de DocumentViewer para mostrar un documento XPS

```
<Window x:Class="WPF_Documentos.Visualizando_XPS"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       Title="Visor de XPS">
    <Grid>
        <DocumentViewer Name="viewer" />
    </Grid>
```

```
</Window>
```

En el code behind se puede cargar el documento usando la clase XpsDocument del espacio de nombres System.Windows.Xps.Packaging, como en el Listado 18- 25.

Listado 18- 25 Carga de un documento XPS desde el code-behind y visualización en un DocumentViewer

```
public Visualizando_XPS()  
{  
    InitializeComponent();  
    XpsDocument document = new XpsDocument(  
        "Lección 7 Figuras.xps",  
        System.IO.FileAccess.Read);  
    viewer.Document = document.GetFixedDocumentSequence();  
}
```

El método GetFixedDocumentSequence retorna una instancia de FixedDocumentSequence que representa una secuencia de FixedDocuments. Esta clase implementa a su vez la interfaz IDocumentPaginatorSource que es el tipo de la propiedad Document de DocumentViewer.

Al desplegar este ejemplo se obtiene algo similar a la Figura 18- 37.



Figura 18- 37 Visor de FixedDocument y documentos XPS

Este visor permite, entre otras funciones, imprimir el documento, copiar un contenido al clipboard, aumentar y disminuir la vista del documento, buscar un texto en el documento..

El botón de imprimir, arriba a la izquierda de la ventana, muestra el diálogo de impresión de Windows como el de la Figura 18- 38. Este dependerá de la versión de Windows que tenga instalada y de las impresoras a las que tenga acceso desde su sistema.

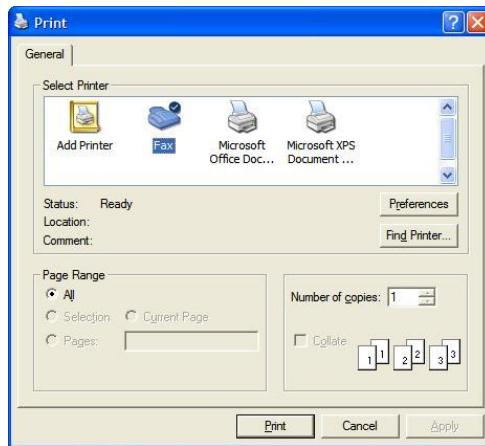


Figura 18- 38 Diálogo de impresión de Windows. Se puede acceder a él desde el visor de documentos XPS de WPF.

La funcionalidad de búsqueda de texto es similar a la de los visores de documentos fluidos vistos antes en esta lección, así como la posibilidad de aumentar y disminuir el tamaño de la visibilidad.

Por último los botones de modo permiten configurar el visor para adaptar el documento a sus dimensiones y para escoger si se deben visualizar una o dos páginas. La Figura 18-39 muestra el documento en cada uno de los cuatro modos de configuración del visor.

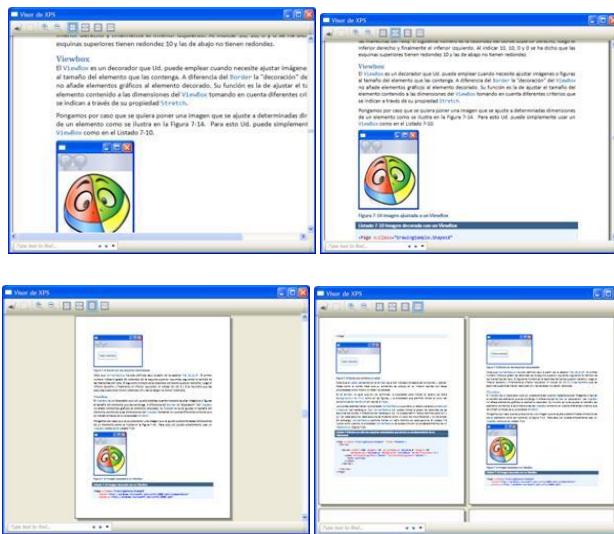


Figura 18- 39 Diferentes modos de visualizar un documento XPS.