WPF

Урок №4

Создание пользовательских элементов управления, Работа с графикой

Содержание

1. Рассмотрение создания пользовательского элемента управления
2. Рассмотрение триггеров
3. Использование пользовательских WPF – элементов управления в Windows Forms приложениях.
4. Практический пример пользовательского элемента управления.
5. Работа с графикой.
6. Фигуры.
7. Практический пример использования
8. Трансформации
9. Отображение мультимедийной информации с помощью MediaElement

1. Рассмотрение создания пользовательского элемента управления

Чтобы объединить функциональные возможности одного или нескольких элементов управления Windows Presentation Foundation (WPF) с пользовательским кодом, можно создать пользовательский элемент управления, также называемый составным элементом управления. Пользовательские элементы управления объединяют быструю разработку элемента управления, стандартные возможности элемента управления WPF и разнообразие пользовательских свойств и методов. В начале создания пользовательского элемента управления предоставляется визуальный конструктор, в котором можно разместить стандартные элементы управления WPF. Дочерние элементы управления называются составляющими элемента управления.

Эти составляющие элемента управления сохраняют все свои функции, а также внешний вид и поведение стандартного элемента управления WPF. После того как эти элементы управления будут встроены в пользовательский элемент управления, они не будут больше доступны через код. Пользовательский элемент управления обладает своим собственным оформлением, а также обрабатывает все основные функции, связанные с элементами управления.

Расширяемость модели элементов управления Windows Presentation Foundation (WPF) значительно уменьшает необходимость создания новых элементов управления. Однако в некоторых случаях разработчику по-прежнему может понадобиться создать пользовательский элемент управления.

WPF предоставляет три общих модели для создания элементов управления, каждая из которых имеет собственный набор функций и уровень гибкости. Основными классами для этих моделей являются UserControl, Control и FrameworkElement.

Самым простым способом создания элемента управления в WPF является создание производной от UserControl. При построении элемента управления, наследующего от UserControl, разработчик добавляет существующие компоненты к UserControl, называет их и ссылается на обработчики событий в Язык XAML (Extensible Application Markup Language). Это позволяет впоследствии ссылаться на именованные элементы и определять обработчики событий в коде. Эта модель разработки очень схожа с моделью, используемой для разработки приложений в WPF:

Будучи построен правильно, UserControl может пользоваться преимуществами расширенного содержимого, стилями и триггерами. В то же самое время, если элемент управления наследует от UserControl, то пользователи элемента управления не смогут применить DataTemplate или ControlTemplate для настройки его внешнего вида. Для создания пользовательского элемента управления, поддерживающего шаблоны, необходимо создать производную от класса Control или от одной из его производных (отличной от UserControl).

Модель, порождаемая от класса Control, используется большинством существующих элементов управления WPF. Когда создается элемент управления, наследующий от класса Control, то его внешний вид определяется при помощи шаблонов. Это позволяет отделить рабочую логику от визуального представления. Разделение пользовательского интерфейса и логики также достигается путем применения команд и привязок вместо событий, а также сокращением использования ссылок на элементы в ControlTemplate. Если пользовательский интерфейс и логика элемента управления должным образом разделены, пользователи элемента могут переопределить ControlTemplate элемента для изменения его внешнего вида. Хотя создание пользовательского Control не является такой же простой процедурой, как создание UserControl, пользовательский Control предоставляет наибольшую гибкость.

Элементы управления, производные от UserControl или от Control, основываются на сочетании существующих элементов. Для многих сце-

нариев это решение приемлемо, так как любой объект, наследующий от FrameworkElement, может быть в ControlTemplate. В то же время бывают случаи, когда для внешнего вида элемента управления требуется функциональность, превосходящая возможности построения простого элемента. В подобных сценариях необходимо строить компоненты на основе FrameworkElement.

Существует два стандартных метода для построения компонентов, основанных на FrameworkElement: прямая отрисовка и настраиваемое построение элемента. Прямая отрисовка включает переопределение метода OnRender из FrameworkElement и предоставление операций DrawingContext, которые явно определяют графические параметры компонента. Этот метод используется Image и Border. Пользовательское построение элемента включает в себя создание экземпляров и использование объектов типа Visual для построения внешнего вида компонента. Пример находится в разделе Использование объектов DrawingVisual. Track является примером элемента управления WPF, который использует пользовательское построение элемента. Также в одном элементе можно комбинировать прямую отрисовку и настраиваемое построение.

Одной из наиболее мощных функций WPF является возможность изменения вида и поведения элемента управления за рамками возможного с помощью основных свойств без необходимости создания нового пользовательского элемента. Стили, привязки данных и триггеры предоставляются системой свойств WPF и системой событий WPF. Если в элементе управления реализуются свойства зависимостей и перенаправленные события, пользователи элемента могут использовать эти функции точно так же, как и для любого элемента управления, поставляемого с WPF, вне зависимости от модели, с помощью которой создается данный элемент.

Если свойство является свойством зависимостей, разработчик может выполнить следующие действия:

- Задать свойство в стиле.

- Привязать свойство к источнику данных.
- Использовать динамический ресурс в качестве значения свойства.
- Анимировать свойство.

Создание DependencyProperty

Если свойство элемента управления должно поддерживать одну из перечисленных функций, необходимо реализовать его как свойство зависимостей. В следующем примере определяется свойство зависимостей, называемое Value, с помощью следующих действий:

1. Определите идентификатор DependencyProperty, называемый ValueProperty, в качестве поля public static readonly.

2. Зарегистрируйте имя свойства в системе свойств путем вызова DependencyProperty.Register, указывая следующие данные:

- Имя свойства.
- Тип свойства.
- Тип владельца свойства.
- Метаданные для свойства. Метаданные содержат значение свойства по умолчанию, CoerceValueCallback и PropertyChangedCallback.

3. Определите свойство-оболочку CLR, называемое Value (это имя совпадает с именем, используемым для регистрации свойства зависимостей), путем реализации методов доступа get и set свойства. Обратите внимание, что методы доступа get и set вызывают только GetValue и SetValue соответственно. Рекомендуется, чтобы методы доступа к свойствам зависимостей не содержали дополнительной логики, так как клиенты и WPF могут пропускать методы доступа и вызывать GetValue и SetValue напрямую. Например, если свойство привязано к источнику данных, то метод доступа свойства set не вызывается. Вместо добавления дополнительной логики к методам доступа получения и установки

следует использовать делегаты ValidateValueCallback, CoerceValueCallback, и PropertyChangedCallback для ответа на значения или их проверки при изменении. Дополнительные сведения об этих обратных вызовах см. в разделе Проверка и обратные вызовы свойства зависимостей.

4. Определите метод для CoerceValueCallback, называемый CoerceValue. CoerceValue позволяет убедиться в том, что Value больше или равно MinValue и меньше или равно MaxValue.

5. Определите метод для PropertyChangedCallback, называемый OnValueChanged. OnValueChanged, который создает объект RoutedPropertyChangedEventArgs(T) и подготавливает создание перенаправленного события ValueChanged.

```
/// <summary>
/// Определяет свойство зависимостей.
/// </summary>
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
        "Value", typeof(decimal), typeof(NumericUpDown),
        new FrameworkPropertyMetadata(MinValue, new
PropertyChangedCallback(OnValueChanged),
                                      new
CoerceValueCallback(CoerceValue)));

/// <summary>
/// Получает или устанавливает значение.
/// </summary>
public decimal Value
{
    get { return (decimal)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}

private static object CoerceValue(DependencyObject element, object
value)
{
```

```
            decimal newValue = (decimal)value;
            NumericUpDown control = (NumericUpDown)element;

            newValue = Math.Max(MinValue, Math.Min(MaxValue, newValue));

            return newValue;
        }

        private static void OnValueChanged(DependencyObject obj,
        DependencyPropertyChangedEventArgs args)
        {
            NumericUpDown control = (NumericUpDown)obj;

            RoutedPropertyChangedEventArgs<decimal> e = new
        RoutedPropertyChangedEventArgs<decimal>(
                (decimal)args.OldValue, (decimal)args.NewValue,
        ValueChangedEvent);
            control.OnValueChanged(e);
        }
```

UserControl

Элементы управления в Windows Presentation Foundation (WPF) поддерживают содержимое, стили, триггеры и шаблоны. Во многих случаях эти возможности позволяют создавать целостное настраиваемое оформление без создания нового элемента управления.

Если вам действительно необходим новый элемент управления, то простейший способ — создать класс, производный от UserControl. Перед этим учтите, что ваш элемент управления не будет поддерживать шаблоны и, следовательно, не будет поддерживать сложные настройки. Однако, извлечение из UserControl является подходящей моделью, если вы хотите построить свое управление, добавляя к нему существующие элементы управления, так же как вы строите приложение, и если у вас нет необходимости поддерживать сложные настройки. (Если вы хотите использовать шаблоны с вашим элементом управления, то вместо этого извлеките из Control.)

UserControlявляется типом ContentControl. Его свойство содержимого представляет собой свойство Content. Свойство Content принадлежит типу Object, поэтому нет ограничений того, что можно поместить в объект ContentControl. Можно использовать Язык XAML или код для установки свойства Content.

Модель содержимого ContentControl используют следующие элементы управления: Button, ButtonBase, CheckBox, ComboBoxItem, ContentControl, Frame, GridViewColumnHeader, GroupItem, Label, ListBoxItem, ListViewItem, NavigationWindow, RadioButton, RepeatButton, ScrollViewer, StatusBarItem, ToggleButton, ToolTip, UserControl, Window.

Создание проекта пользовательского элемента управления в WPF

Запустите версию Visual Studio, которая поддерживает разработку WPF, например Visual Studio 2008.

В меню Файл наведите указатель на пункт Создать и выберите Проект. Откроется диалоговое окно Создать проект.

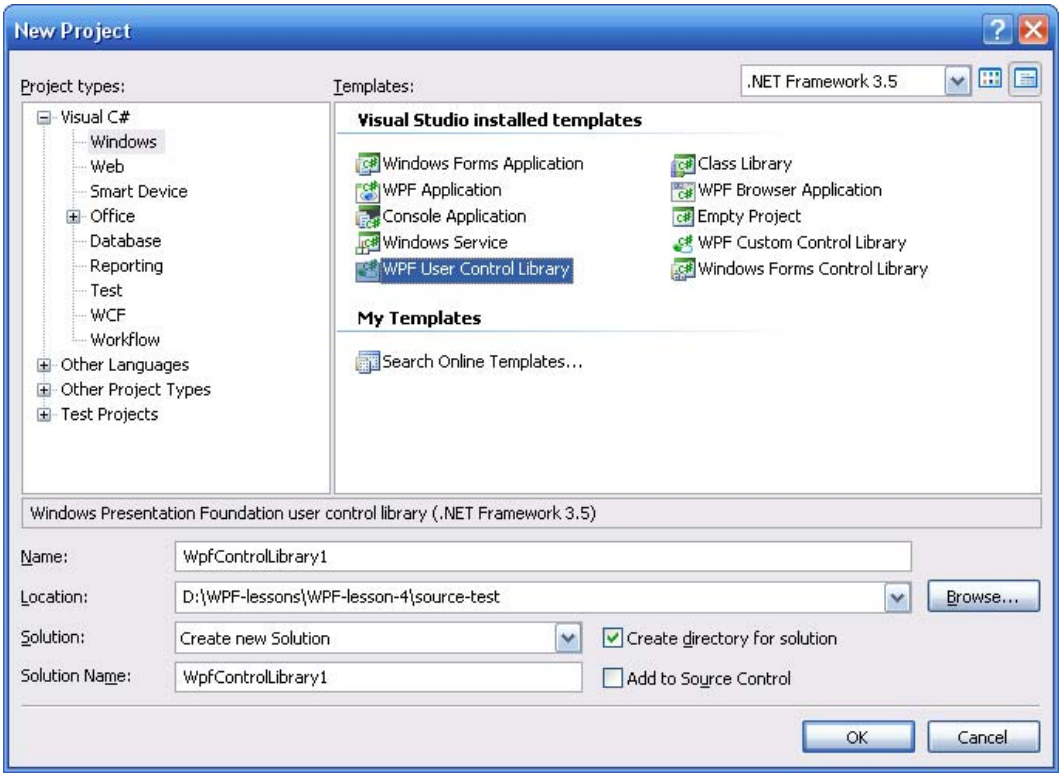
В области Типы проектов выберите язык программирования, который предполагается использовать (C#).

В области Шаблоны выберите Библиотека пользовательских элементов управления WPF (WPF user Control Library) для проектов Visual Basic или Visual C#. Дополнительные сведения о выбранном шаблоне отображаются под областями Типы проектов и Шаблоны.

В текстовом поле Имя присвойте проекту уникальное имя, соответствующее назначению элемента управления.

В поле Расположение введите каталог для сохранения проекта или нажмите кнопку Обзор для перехода к требуемому каталогу.

Нажмите кнопку ОК. Откроется Windows Presentation Foundation для Visual Studio (конструктор), в котором отображается элемент управления «UserControl1» созданного проекта.



Перетащите элементы управления с Панели элементов на пользовательский элемент управления.

Эти элементы управления должны быть размещены и спроектированы сразу так, как они будут выглядеть в конечном пользовательском элементе управления. Если нужно разрешить разработчикам доступ к элементам управления, составляющим новый элемент, необходимо объявить их как открытые или же выборочно представить свойства этих элементов управления.

2. Рассмотрение триггеров

Классы Style, ControlTemplate и DataTemplate, которые помогают изменить внешний вид элемента управления, имеют свойство Triggers, которое может содержать набор триггеров. Триггер задает свойства или начинает действия (например, анимацию) при изменении значения свойства или при возникновении события.

Триггеры свойств

Чтобы продемонстрировать использование триггеров для установки свойств, сделаем каждый ListBoxItem частично прозрачным, если он не является выбранным.

Следующий стиль устанавливает значение Opacity элемента ListBoxItem равным 0.5. Когда свойство IsSelected равно true, Opacity устанавливается в 1.0.

```
<Style TargetType="ListBoxItem">
  <Setter Property="Opacity" Value="0.5" />
  <Setter Property="MaxHeight" Value="75" />
  <Style.Triggers>
    <Trigger Property="IsSelected" Value="True">
        <Setter Property="Opacity" Value="1.0" />
    </Trigger>


...


  </Style.Triggers>
</Style>
```

В этом примере Trigger используется для установки значения свойства. Обратите внимание, что класс Trigger также имеет свойства EnterActions и ExitActions, которые позволяют триггеру выполнять действия.

Триггеры свойств

В предыдущем примере было показано, что Trigger задает значения свойств или запускает действия, основанные на значении свойства. Другим типом триггера является EventTrigger, запускающий набор действий в зависимости от наличия события. Например, следующие объекты EventTrigger указывают на то, что при вхождении указателя мыши в ListBoxItem свойство MaxHeight изменяется до значения 90 за период в 0.2 секунды. Когда указатель мыши перемещается из элемента, свойство возвращается к исходному значению за 1 секунду. Обратите внимание, что значение To не обязательно указывать для анимации MouseLeave. Это происходит потому, что анимация имеет возможность следить за исходным значением.

```
<EventTrigger RoutedEvent="Mouse.MouseEnter">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation
          Duration="0:0:0.2"
          Storyboard.TargetProperty="MaxHeight"
          To="90"  />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Mouse.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation
          Duration="0:0:1"
          Storyboard.TargetProperty="MaxHeight"  />
      </Storyboard>
    </BeginStoryboard>
```

```
    </EventTrigger.Actions>
  </EventTrigger>
```

Триггеры данных

В дополнение к Trigger и EventTrigger существуют другие типы триггеров. MultiTrigger позволяет задавать значения свойств в зависимости от нескольких условий. DataTrigger и MultiDataTrigger используются в том случае, когда свойство условия связано с данными.

В следующем примере ItemsSource, принадлежащий ListBox привязан к Адресам, ObservableCollection<(Of <(T>)>) объектов Адреса. Объекты Адреса имеют свойства Имя и Состояние. Определения параметров Place и Places не отображаются, но их можно просмотреть в примере (Source\datatrigers).

Каждый элемент ListBoxItem списка ListBox отображает объект Place. Style в данном примере применяется к каждому элементу ListBoxItem.

DataTrigger указан таким образом, что если в качестве Состояния элемента данных Адреса задано значение "WA", то основным цветом соответствующего ListBoxItem является красный.

```
<EventTrigger RoutedEvent="Mouse.MouseEnter">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation
          Duration="0:0:0.2"
          Storyboard.TargetProperty="MaxHeight"
          To="90"  />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

```
<EventTrigger RoutedEvent="Mouse.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation
          Duration="0:0:1"
          Storyboard.TargetProperty="MaxHeight"  />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

В примере каждый элемент ListBoxItem списка ListBox отображает объект Place. Style в данном примере применяется к каждому элементу ListBoxItem. Элементы Condition триггера MultiDataTrigger показывают, что если параметры Name и State элемента данных Place имеют соответственно значения Portland и OR, фон соответствующего элемента ListBoxItem получает значение Cyan.

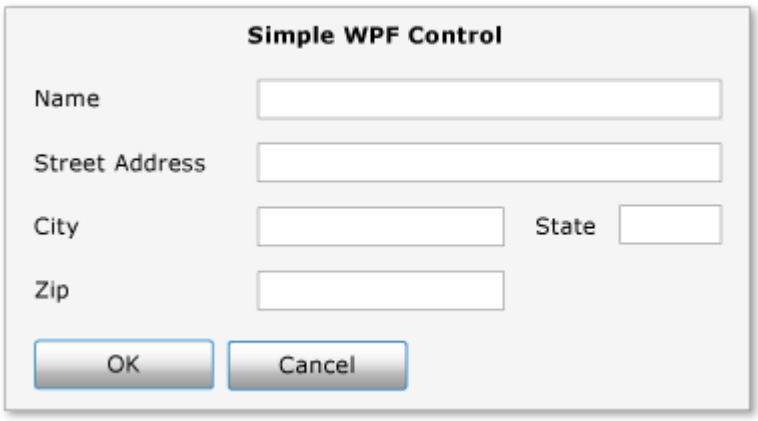
```
<Window.Resources>
  <c:Places x:Key="PlacesData"/>

  <Style TargetType="ListBoxItem">
    <Style.Triggers>
      <DataTrigger Binding="{Binding Path=State}" Value="WA">
        <Setter Property="Foreground" Value="Red" />
      </DataTrigger>
      <MultiDataTrigger>
        <MultiDataTrigger.Conditions>
          <Condition Binding="{Binding Path=Name}" Value="Portland" />
          <Condition Binding="{Binding Path=State}" Value="OR" />
        </MultiDataTrigger.Conditions>
        <Setter Property="Background" Value="Cyan" />
      </MultiDataTrigger>
    </Style.Triggers>
  </Style>
```

```
        <DataTemplate DataType="{x:Type c:Place}">
          <Canvas Width="160" Height="20">
            <TextBlock FontSize="12"
                   Width="130" Canvas.Left="0" Text="{Binding Path=Name}"/>
            <TextBlock FontSize="12" Width="30"
                       Canvas.Left="130" Text="{Binding Path=State}"/>
          </Canvas>
        </DataTemplate>
      </Window.Resources>

      <StackPanel>
        <TextBlock FontSize="18" Margin="5" FontWeight="Bold"
          HorizontalAlignment="Center">Data Trigger Sample</TextBlock>
        <ListBox Width="180" HorizontalAlignment="Center"
      Background="Honeydew"
          ItemsSource="{Binding Source={StaticResource PlacesData}}"/>
      </StackPanel>
```

3. Использование пользовательских WPF – элементов управления в Windows Forms приложениях

Клиент Windows Presentation Foundation (WPF) предоставляет среду с широкими возможностями для создания приложений. Однако если имеющийся код Windows Forms содержит важные элементы, более эффективным может быть расширение существующего приложения Windows Forms с помощью клиента WPF, а не переписывание кода «с нуля». Распространенным случаем является задача внедрить одну или нескольких страниц, реализованных с помощью WPF, в приложение Windows Forms (Source\wpfinwf).

Страница WPF, используемая в этом примере, представляет собой простую форму ввода данных, содержащую имя пользователя и адрес. Когда пользователь нажимает одну из двух кнопок, чтобы указать, что задача завершена, страница создает пользовательское событие для возвращения сведений на узел.



Создание библиотеки с элементами WPF

Запустите приложение Microsoft Visual Studio и откройте диалоговое окно New project.

Выберите шаблон WPF browser application.

Назовите новый проект MyControls и поместите его в папку верхнего уровня с подходящим именем, например — WfHostingWpf. Позже ведущее приложение будет помещено в эту папку. Нажмите кнопку ОК, чтобы создать проект. По умолчанию проект содержит одну страницу с именем Page1.

Щелкните правой кнопкой имя проекта в обозревателе решений и выберите Свойства.

Задайте для параметра Тип выходных данных значение Библиотека классов, чтобы скомпилировать страницу как библиотеку DLL.

Удалите файлы определения приложения MyApp.xaml и MyApp.xaml.cs из проекта. Эти файлы нужны, только если страница реализуется как приложение.

При компиляции приложения WPF как библиотеки классов, его невозможно запустить для просмотра отображаемой страницы. По этой причине, может оказаться более удобным оставить выходной тип «Приложение Windows», пока приложение не будет полностью реализовано. Это позволит проверять внешний вид страниц при запуске приложения. По завершении, удалите файлы определения приложения и измените тип выходных данных на «Библиотека классов», чтобы скомпилировать приложение как библиотеку DLL.

Проект должен иметь ссылки на следующие системные библиотеки DLL: System, PresentationCore, PresentationFramework, WindowsBase. Если какая-либо из этих библиотек DLL не включена по умолчанию, добавьте ее в проект.

Пользовательский интерфейс страницы состоит из пяти элементов TextBox. Каждый элемент TextBox имеет связанный элемент TextBlock, служащий в качестве метки. В нижней части страницы имеются два элемента Button — ОК и Отмена. При нажатии любой кнопки, страница создает пользовательское событие для возвращения сведения на узел.

Файл с выделенным кодом страницы WPF Page1.xaml.cs реализует четыре основных задачи:

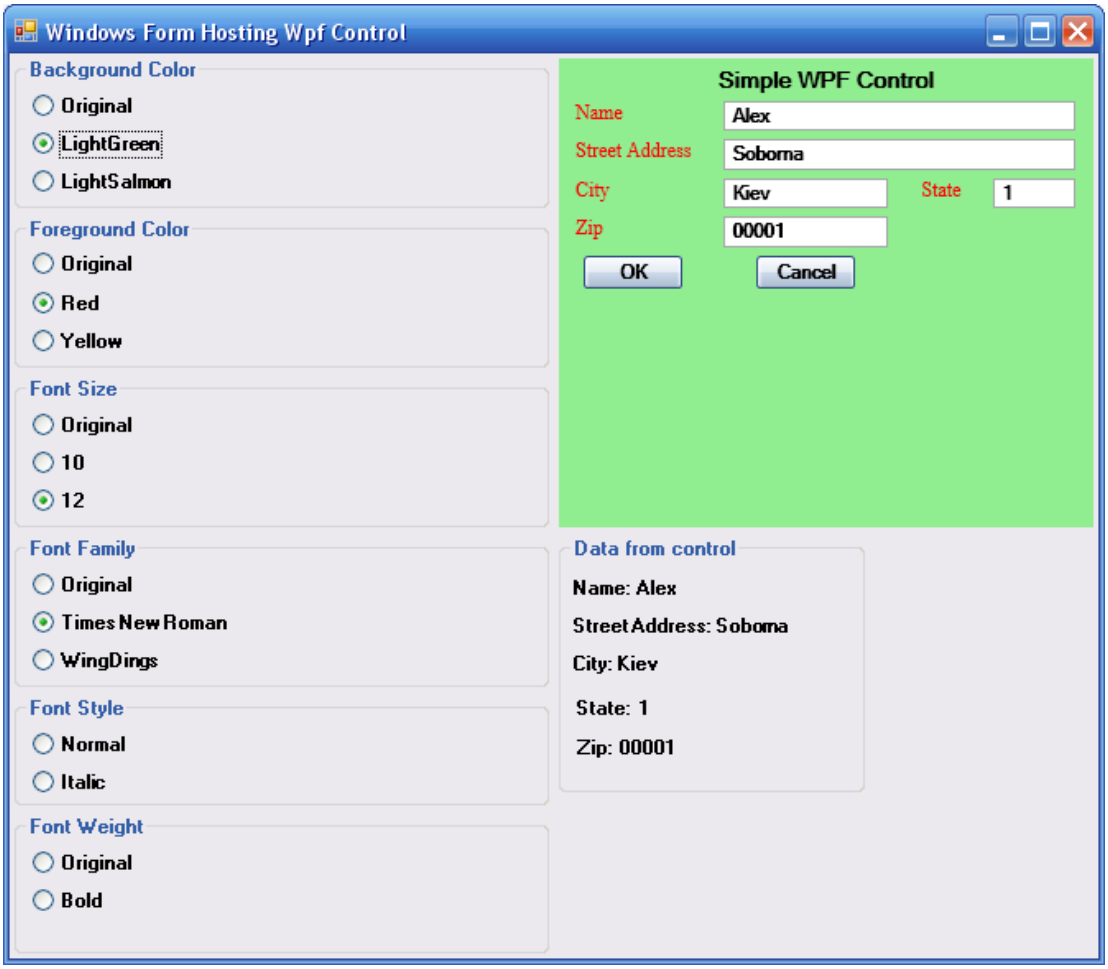
Регистрирует имя библиотеки DLL страницы с объектом Application, чтобы он знал, откуда загружать страницу.

Обрабатывает событие, когда пользователь нажимает одну из кнопок.

Извлекает данные из элементов TextBox и пакует их в объект аргумента пользовательского события.

Вызывает пользовательское событие OnButtonClick, которое уведомляет узел, что пользователь завершил работу, и возвращает данные на узел.

Для размещения страницы WPF в форме ведущее приложение Windows Forms использует объект ElementHost. Для получения данных из формы приложение обрабатывает событие OnButtonClick страницы. Приложение также содержит набор переключателей, которые можно использовать для изменения внешнего вида страницы. На следующем рисунке показана отображаемая форма.



Метод Form1_Load из примера кода показывает общую процедуру размещения элемента управления WPF.

Создайте новый объект ElementHost.

Присвойте свойству Dock элемента управления значение DockStyle..::.Fill.

Добавьте элемент управления ElementHost к коллекции Controls элементов управления Panel.

Создайте экземпляр страницы WPF.

Разместите страницу в форме путем назначения страницы свойству Child элемента управления ElementHost.

4. Практический пример пользовательского элемента управления

Теперь давайте рассмотрим большой практический пример создания элемента управления (Source\numeric). В этом примере демонстрируется создание простого пользовательского элемента управления, который поддерживает шаблоны, с последующей упаковкой этого элемента управления в собственную сборку.

В примере есть два проекта: CustomControlLibrary – библиотека элементов управления, CustomControlProj – приложение которое использует новый компонент.

Код нового элемента управления реализован в файле NumericUpDown.cs. Создается класс NumericUpDown наследник Control.

Для начала в статическом конструкторе регистрируется событие мыши.

```
static NumericUpDown()
        {
            InitializeCommands();

            // Listen to MouseLeftButtonDown event to determine if
slide should move focus to itself
            EventManager.RegisterClassHandler(typeof(NumericUpDown),
                Mouse.MouseDownEvent, new
MouseButtonEventHandler(NumericUpDown.OnMouseLeftButtonDown), true);


DefaultStyleKeyProperty.OverrideMetadata(typeof(NumericUpDown), new
FrameworkPropertyMetadata(typeof(NumericUpDown)));
        }
```

Далее в регионе кода Properties создаются свойства необходимые контролу (#region Properties).

Далее создается событие изменения значения элемента.

```
        #region Events
                /// <summary>
                /// Identifies the ValueChanged routed event.
                /// </summary>
                public static readonly RoutedEvent ValueChangedEvent =
        EventManager.RegisterRoutedEvent(
                    "ValueChanged", RoutingStrategy.Bubble,
                    typeof(RoutedPropertyChangedEventHandler<decimal>),
        typeof(NumericUpDown));

                /// <summary>
                /// Occurs when the Value property changes.
                /// </summary>
                public event RoutedPropertyChangedEventHandler<decimal>
        ValueChanged
                {
                    add { AddHandler(ValueChangedEvent, value); }
                    remove { RemoveHandler(ValueChangedEvent, value); }
                }
                #endregion
```

Также создаются две команды IncreaseCommand и DecreaseCommand.

```
        #region Commands

                public static RoutedCommand IncreaseCommand
                {
                    get
                    {
                        return _increaseCommand;
                    }
                }
                public static RoutedCommand DecreaseCommand
                {
                    get
```

```
            {
                return _decreaseCommand;
            }
        }

        private static void InitializeCommands()
        {
            _increaseCommand = new
RoutedCommand("IncreaseCommand",typeof(NumericUpDown));

CommandManager.RegisterClassCommandBinding(typeof(NumericUpDown), new
CommandBinding(_increaseCommand, OnIncreaseCommand));

CommandManager.RegisterClassInputBinding(typeof(NumericUpDown), new
InputBinding(_increaseCommand, new KeyGesture(Key.Up)));

            _decreaseCommand = new RoutedCommand("DecreaseCommand",
typeof(NumericUpDown));

CommandManager.RegisterClassCommandBinding(typeof(NumericUpDown), new
CommandBinding(_decreaseCommand, OnDecreaseCommand));

CommandManager.RegisterClassInputBinding(typeof(NumericUpDown), new
InputBinding(_decreaseCommand, new KeyGesture(Key.Down)));
        }

        private static void OnIncreaseCommand(object sender,
ExecutedRoutedEventArgs e)
        {
            NumericUpDown control = sender as NumericUpDown;
            if (control != null)
            {
                control.OnIncrease();
            }
        }
        private static void OnDecreaseCommand(object sender,
ExecutedRoutedEventArgs e)
        {
            NumericUpDown control = sender as NumericUpDown;
            if (control != null)
```

```
                {
                    control.OnDecrease();
                }
            }

            protected virtual void OnIncrease()
            {
                this.Value+=Change;
            }
            protected virtual void OnDecrease()
            {
                this.Value-=Change;
            }

            private static RoutedCommand _increaseCommand;
            private static RoutedCommand _decreaseCommand;
            #endregion
```

Таким образом наш компонент имеет такие свойства: Value, Minimum, Maximum, Change, DecimalPlaces, Valustring. Обрабатывает событие ValueChanged. И принимает команды IncreaseCommand и DecreaseCommand.

Внешний вид компонента описан с помощью тем в файлах generic.xaml и Luna.NormalColor.xaml.

Для использования созданного компонента в своем проекте нужно сделать такие действия.

1. Добавить ссылку на сборку с компонентом в список References.

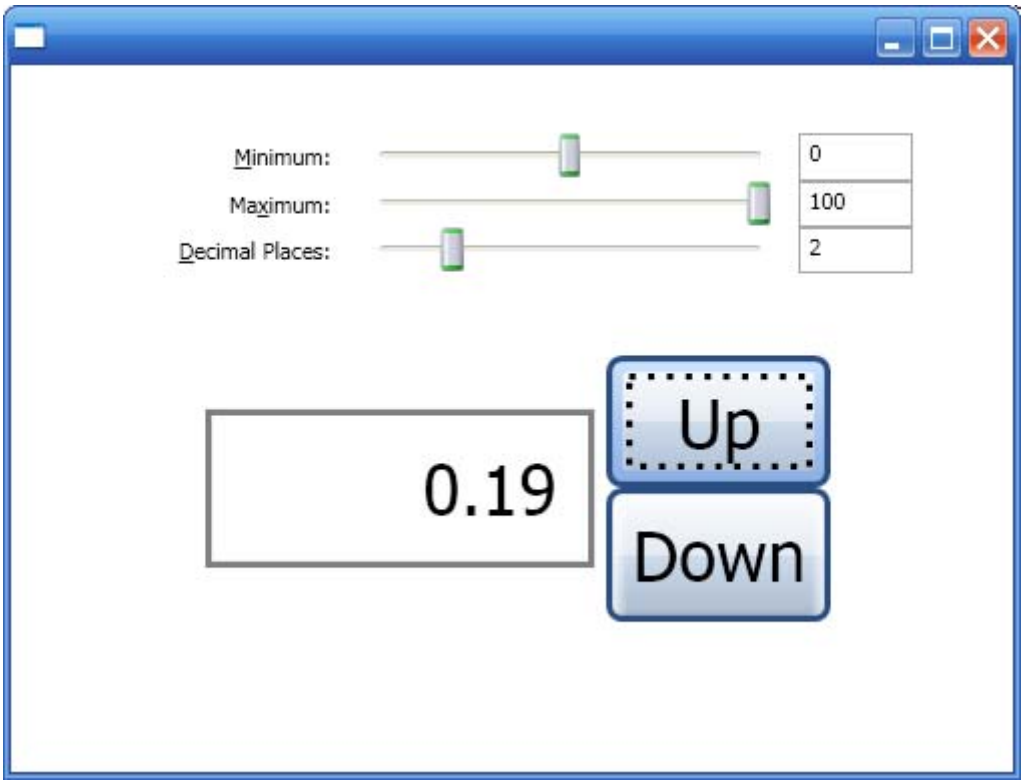
2. Прописать ссылку на сборку в разметке окна.

```
<Window x:Class="SDKSample.NumericUpDown.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:uc="clr-
namespace:CustomControlLibrary;assembly=CustomControlLibrary">
```

При добавлении компонента нужно учитывать имя пространства имен заданное ранее.

```
<Viewbox Grid.Row="3" Grid.ColumnSpan="3" Width="400">
        <uc:NumericUpDown x:Name="nud" Change=".01" Margin="10"/>
    </Viewbox>
```

В результате мы получили такое вот приложение, которое использует новый элемент управления.



5. Работа с графикой

Windows Presentation Foundation включает поддержку высокого качества двухмерный и графики, анимации и мультимедиа трехмерный. Основные возможности графической платформы включают:

Поддержка векторной графики.

Аппаратное ускорение.

Разрешение и аппаратно-независимая графика.

Минимальная перерисовка экрана и интегрированная система анимации.

WPF предоставляет дополнительные графические и анимационные возможности, которые были ранее доступны только из специализированных библиотек — в частности, Интерфейс графических устройств (GDI) Microsoft Windows и Microsoft Windows GDI+. WPF предлагает встроенную поддержку мультимедиа, векторной графики, анимации и композиции содержимого, упрощая разработчикам создание оригинального пользовательского интерфейса и содержимого. Используя Microsoft Visual Studio .NET или даже текстовый редактор, такой как Блокнот (Майкрософт), можно создать векторную графику или сложную анимацию и интегрировать мультимедиа в приложения.

WPF предоставляет библиотеку, часто используемую для векторного рисования фигур, таких как прямоугольники и эллипсы. Эти встроенные фигуры WPF являются не просто фигурами. Они являются элементами программирования, которые реализуют многие возможности других наиболее распространенных элементов управления, включая клавиатуру и мышь.

Используя эффекты анимации, можно заставить элементы управления увеличиваться, вибрировать, вращаться или растворяться, а также создать оригинальные переходы между страницами и т.д. Поскольку WPF позволяет анимировать большинство свойств, можно не только анимировать большинство объектов WPF, но также использовать WPF для анимации создаваемых пользователем объектов.

Эффекты для точечных рисунков

Эффекты точечного рисунка позволяют конструкторам и разработчикам применять визуальные эффекты к отображаемому содержимому WPF (Windows Presentation Foundation). Например, растровые эффекты позволяют легко применить эффект DropShadowBitmapEffect или эффект размытия к изображению или кнопке.

Эффекты точечного рисунка (объект BitmapEffect) являются операциями обработки простой точки. Эффект точечного рисунка принимает BitmapSource в качестве входных данных и создает новый BitmapSource после применения эффекта, такого как размытой или отброшенной тени. Каждый эффект точечного рисунка предоставляет свойства, которые могут управлять свойствами фильтрации, такими как Radius из BlurBitmapEffect.

В качестве особого случая в WPF эффекты могут задаваться как свойства для объектов в режиме реального времени Visual, таких как Button или TextBox. Обработка точки применяется и отображается во время выполнения. В этом случае во время отрисовки, Visual автоматически преобразуется в свой эквивалент BitmapSource и подается в качестве входных данных для BitmapEffect. Выходные данные заменяют поведение при отрисовке по умолчанию Visual объекта. Поэтому объекты BitmapEffect заставляют визуальные изображения отображаться только в программном обеспечении, т.е. эффекты на визуальных изображениях применяются без аппаратного ускорения.

BlurBitmapEffect имитирует просмотр объекта через расфокусированную лупу.

OuterGlowBitmapEffect создает цветное сияние вокруг объекта.

DropShadowBitmapEffect создает тень позади объекта.

BevelBitmapEffect создает скос, который приподнимает поверхность изображения в соответствии с указанной кривой.

EmbossBitmapEffect создает отображение элементов рельефа из Visual для создания впечатления текстуры и глубины от искусственного источника света.

BitmapEffect является свойством Visual. Таким образом, применение эффектов к Visuals, такие как Button, Image, DrawingVisual или UIElement, является таким же простым как задание свойства.BitmapEffect может быть установлено на единственный объект BitmapEffect или несколько эффектов могут быть связанными при помощи объекта BitmapEffectGroup.

В следующем примере демонстрируется применение BitmapEffect.

```
<Button  Width="200">You Can't Read This!
  <Button.BitmapEffect>
    <BlurBitmapEffect Radius="10" KernelType="Box" />

  </Button.BitmapEffect>
</Button>
```

В следующем примере демонстрируется применение BitmapEffect в коде.

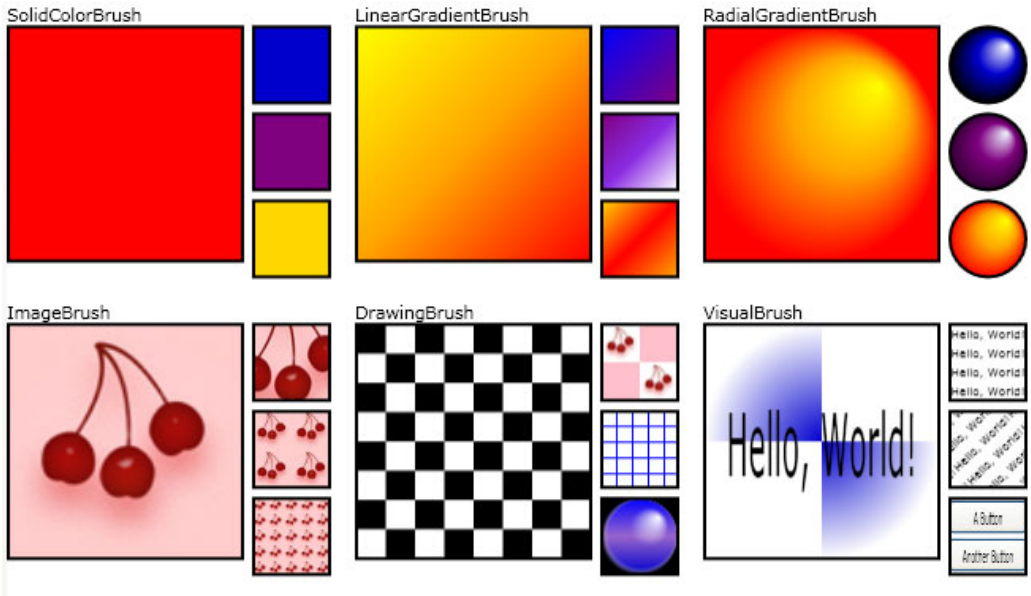
```
// Get a reference to the Button.
Button myButton = (Button)sender;

// Initialize a new BlurBitmapEffect that will be applied
// to the Button.
BlurBitmapEffect myBlurEffect = new BlurBitmapEffect();

// Set the Radius property of the blur. This determines how
// blurry the effect will be. The larger the radius, the more
// blurring.
myBlurEffect.Radius = 10;

// Set the KernelType property of the blur. A KernalType of "Box"
```

```
// creates less blur than the Gaussian kernal type.
myBlurEffect.KernelType = KernelType.Box;

// Apply the bitmap effect to the Button.
myButton.BitmapEffect = myBlurEffect;
```

Кисти

Все отображаемые на экране объекты являются видимыми, так как они нарисованы кистью. Например, кисть используется для описания фона кнопки, основного цвета текста и заполнения фигуры. В этом разделе представлены механизмы рисования с помощью кистей Windows Presentation Foundation и примеры. Кисти позволяют рисовать как простые одноцветные объекты пользовательский интерфейс, так и объекты со сложными наборами шаблонов и изображений.

Brush «рисует» область с выходными данными. Различные кисти имеют разные типы вывода. Некоторые кисти закрашивают область сплошным цветом, другим — градиентом, узором, изображением или рисунком. На следующем рисунке приведены примеры каждого из типов Brush.



SolidColorBrush закрашивает область сплошным Color. Существует несколько способов указания ColorSolidColorBrush: например, можно

указать его альфа-канал, красный, синий и зеленый каналы или воспользоваться одним из цветов, предоставленных классом Colors.

В следующем примере SolidColorBrush используется для закрашивания Fill фигуры Rectangle.

```
<Rectangle Width="75" Height="75">
  <Rectangle.Fill>
    <SolidColorBrush Color="Red" />
  </Rectangle.Fill>
</Rectangle>
```

LinearGradientBrush закрашивает область линейным градиентом. Линейный градиент сочетает два или более цветов через линию, являющуюся осью градиента. Используйте объекты GradientStop для указания цветов градиента и их позиции.

В следующем примере LinearGradientBrush используется для закрашивания Fill фигуры Rectangle.

```
<Rectangle Width="75" Height="75">
  <Rectangle.Fill>
    <LinearGradientBrush>
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="Orange" Offset="0.5" />
      <GradientStop Color="Red" Offset="1.0" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

RadialGradientBrush закрашивает область радиальным градиентом. Радиальный градиент сочетает несколько цветов через окружность. Как и при использовании класса LinearGradientBrush, необходимо использовать объекты GradientStop для указания цветов в градиенте и их позиций.

В следующем примере RadialGradientBrush используется для закрашивания Fill фигуры Rectangle.

```
<Rectangle Width="75" Height="75">
  <Rectangle.Fill>
    <RadialGradientBrush GradientOrigin="0.75,0.25">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="Orange" Offset="0.5" />
      <GradientStop Color="Red" Offset="1.0" />
    </RadialGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

ImageBrush закрашивает область с помощью ImageSource. В следующем примере используется ImageBrush для закрашивания Fill объекта Rectangle.

```
<Rectangle Width="75" Height="75">
  <Rectangle.Fill>
    <ImageBrush ImageSource="sampleImages\pinkcherries.jpg"  />
  </Rectangle.Fill>
</Rectangle>
```

DrawingBrush закрашивает область с помощью Drawing. Drawing может содержать фигуры, изображения, текст и мультимедийные данные. В следующем примере DrawingBrush используется для закрашивания Fill фигуры Rectangle.

```
<Rectangle Width="75" Height="75">
  <Rectangle.Fill>
    <DrawingBrush Viewport="0,0,0.25,0.25" TileMode="Tile">
      <DrawingBrush.Drawing>
        <DrawingGroup>
          <GeometryDrawing Brush="White">
            <GeometryDrawing.Geometry>
              <RectangleGeometry Rect="0,0,100,100" />
```

```
            </GeometryDrawing.Geometry>
          </GeometryDrawing>

          <GeometryDrawing>
            <GeometryDrawing.Geometry>
              <GeometryGroup>
                <RectangleGeometry Rect="0,0,50,50" />
                <RectangleGeometry Rect="50,50,50,50" />
              </GeometryGroup>
            </GeometryDrawing.Geometry>
            <GeometryDrawing.Brush>
              <LinearGradientBrush>
                <GradientStop Offset="0.0" Color="Black" />
                <GradientStop Offset="1.0" Color="Gray" />
              </LinearGradientBrush>
            </GeometryDrawing.Brush>
          </GeometryDrawing>
        </DrawingGroup>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Rectangle.Fill>
</Rectangle>
```

VisualBrush закрашивает область с помощью объекта Visual. В число объектов Visual входят объекты Button, Page и MediaElement. VisualBrush также позволяет переносить содержимое из одной части приложения в другую область, что может быть полезным при создании эффекта отражения и увеличения части экрана. В следующем примере VisualBrush используется для закрашивания Fill фигуры Rectangle.

```
<Rectangle Width="75" Height="75">
  <Rectangle.Fill>
    <VisualBrush TileMode="Tile">
      <VisualBrush.Visual>
        <StackPanel>
          <StackPanel.Background>
            <DrawingBrush>
              <DrawingBrush.Drawing>
```

```
                    <GeometryDrawing>
                      <GeometryDrawing.Brush>
                        <RadialGradientBrush>
                          <GradientStop Color="MediumBlue" Offset="0.0" />
                          <GradientStop Color="White" Offset="1.0" />
                        </RadialGradientBrush>
                      </GeometryDrawing.Brush>
                      <GeometryDrawing.Geometry>
                        <GeometryGroup>
                          <RectangleGeometry Rect="0,0,50,50" />
                          <RectangleGeometry Rect="50,50,50,50" />
                        </GeometryGroup>
                      </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                  </DrawingBrush.Drawing>
                </DrawingBrush>
              </StackPanel.Background>
              <TextBlock FontSize="10pt" Margin="10">Hello,
    World!</TextBlock>
            </StackPanel>
          </VisualBrush.Visual>
        </VisualBrush>
      </Rectangle.Fill>
    </Rectangle>
```

Рисование через Drawing

В этом разделе представлены объекты Drawing и инструкции по их использованию для эффективного рисования фигур, точечных рисунков, текста и мультимедиа. Используйте объекты Drawing при создании картинок, рисования при помощи DrawingBrush или использовании объектов Visual.

Объект Drawing описывает отображаемое содержимое, такое как фигура, точечный рисунок, видео или строка текста. Различные типы графических объектов описывают различные типы содержимого. Ниже приведен список различных типов графических объектов.

GeometryDrawing — выводит фигуру.

ImageDrawing — выводит изображение.

GlyphRunDrawing — выводит текст.

VideoDrawing — воспроизводит аудио- или видеофайл.

DrawingGroup — выводит другие объекты Drawing. Используйте группировку изображений для объединения других рисунков в один составной рисунок.

Объекты Drawing являются универсальными. Существует множество способов использования объекта Drawing.

Можно отобразить его в виде изображения с помощью DrawingImage и элемента управления Image.

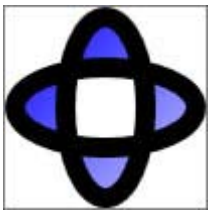
Можно использовать его с DrawingBrush для рисования объекта, например, Background для Page.

Можно использовать его для описания внешнего вида DrawingVisual.

Можно использовать его для перечисления содержимого Visual.

Чтобы нарисовать фигуру, используется объект GeometryDrawing. Его свойство Geometry описывает фигуру, которую нужно нарисовать, его свойство Brush описывает способ закраски внутренней части фигуры, его свойство Pen описывает способ прорисовки его структуры.

В следующем примере GeometryDrawing используется для рисования фигуры. Фигура описывается GeometryGroup и двумя объектами EllipseGeometry. Внутренняя часть фигуры рисуется с помощью LinearGradientBrush, а ее контур — с помощью Black Pen.

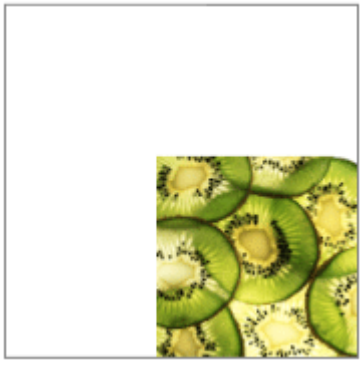


```
<GeometryDrawing>
  <GeometryDrawing.Geometry>

    <!-- Create a composite shape. -->
```

```
        <GeometryGroup>
          <EllipseGeometry Center="50,50" RadiusX="45" RadiusY="20" />
          <EllipseGeometry Center="50,50" RadiusX="20" RadiusY="45" />
        </GeometryGroup>
      </GeometryDrawing.Geometry>
      <GeometryDrawing.Brush>

        <!-- Paint the drawing with a gradient. -->
        <LinearGradientBrush>
          <GradientStop Offset="0.0" Color="Blue" />
          <GradientStop Offset="1.0" Color="#CCCCFF" />
        </LinearGradientBrush>
      </GeometryDrawing.Brush>
      <GeometryDrawing.Pen>

        <!-- Outline the drawing with a solid color. -->
        <Pen Thickness="10" Brush="Black" />
      </GeometryDrawing.Pen>
    </GeometryDrawing>
```

Для рисования изображения используется ImageDrawing. Свойство объекта ImageDrawingImageSource описывает изображение, которое нужно нарисовать, а его свойство Rect определяет область, где рисуется изображение.

В следующем примере изображение рисуется в прямоугольнике, расположенном в точке (75,75), размером 100 на 100 точек. На следующем рисунке показан ImageDrawing, созданный в примере. Серая граница была добавлена, чтобы показать границы ImageDrawing.



```
<!-- The Rect property specifies that the image only fill a 100 by 100
     rectangular area. -->
<ImageDrawing Rect="75,75,100,100"
ImageSource="sampleImages\kiwi.png"/>
```

В следующем примере VideoDrawing и MediaPlayer используются для однократного воспроизведения видеофайла.

```
//
// Create a VideoDrawing.
//
MediaPlayer player = new MediaPlayer();
player.Open(new Uri(@"sampleMedia\xbox.wmv", UriKind.Relative));
VideoDrawing aVideoDrawing = new VideoDrawing();
aVideoDrawing.Rect = new Rect(0, 0, 100, 100);
aVideoDrawing.Player = player;
// Play the video once.
player.Play();
```

Чтобы вывести текст, используется GlyphRunDrawing и GlyphRun. В следующем примере GlyphRunDrawing используется для вывода текста «Hello World».

```
<GlyphRunDrawing ForegroundBrush="Black">
  <GlyphRunDrawing.GlyphRun>
    <GlyphRun
      CaretStops="{x:Null}"
      ClusterMap="{x:Null}"
      IsSideways="False"
      GlyphOffsets="{x:Null}"
      GlyphIndices="43 72 79 79 82 3 58 82 85 79 71"
      BaselineOrigin="0,12.29"
      FontRenderingEmSize="13.333333333333334"
      DeviceFontName="{x:Null}"
      AdvanceWidths="9.62666666666667 7.41333333333333 2.96 2.96
7.41333333333333 3.70666666666667 12.5866666666667 7.41333333333333
```

```
      4.44 2.96 7.41333333333333"
            BidiLevel="0">
            <GlyphRun.GlyphTypeface>
              <GlyphTypeface FontUri="C:\WINDOWS\Fonts\TIMES.TTF" />
            </GlyphRun.GlyphTypeface>
          </GlyphRun>
        </GlyphRunDrawing.GlyphRun>
      </GlyphRunDrawing>
      // Play the video once.
      player.Play();
```

6. Фигуры

Windows Presentation Foundation предоставляет несколько уровней доступа к изображениям и службам визуализации. На верхнем слое объекты Shape просты в использовании и предоставляют множество полезных возможностей, таких как разметка и участие в системе событий Windows Presentation Foundation.

WPF предоставляет ряд готовых к использованию объектов Shape. Все объекты фигур наследуются из класса Shape. Доступны объекты фигур Ellipse, Line, Path, Polygon, Polyline и Rectangle. Объекты Shape используют следующие общие свойства.

Stroke: описывает способ рисования контура фигуры.

StrokeThickness: описывает толщину контура фигуры.

Fill: описывает способ закрашивания внутренней части фигуры.

Свойства данных, определяющие координаты и вершины, измеряемые в аппаратно независимых точках.

Поскольку объекты фигур являются производными класса UIElement, они могут использоваться внутри панелей и большинства элементов управления. Панель Canvas особенно подходит для создания сложных рисунков, поскольку она поддерживает абсолютное позиционирование дочерних объектов.

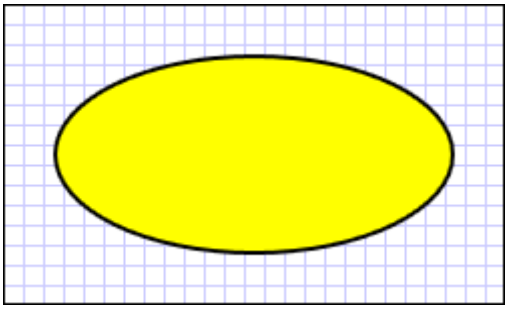
Класс Line позволяет нарисовать линию между двумя точками. В следующем примере показано несколько способов указания координат линии и свойств штриха.

```
<Canvas Height="300" Width="300">

  <!-- Draws a diagonal line from (10,10) to (50,50). -->
  <Line
    X1="10" Y1="10"
    X2="50" Y2="50"
```

```
        Stroke="Black"
        StrokeThickness="4" />

      <!-- Draws a diagonal line from (10,10) to (50,50)
           and moves it 100 pixels to the right. -->
      <Line
        X1="10" Y1="10"
        X2="50" Y2="50"
        StrokeThickness="4"
        Canvas.Left="100">
        <Line.Stroke>
          <RadialGradientBrush GradientOrigin="0.5,0.5" Center="0.5,0.5"
    RadiusX="0.5" RadiusY="0.5">
            <RadialGradientBrush.GradientStops>
              <GradientStop Color="Red" Offset="0" />
              <GradientStop Color="Blue" Offset="0.25" />
            </RadialGradientBrush.GradientStops>
          </RadialGradientBrush>
        </Line.Stroke>
      </Line>

      <!-- Draws a horizontal line from (10,60) to (150,60). -->
      <Line
         X1="10" Y1="60"
         X2="150" Y2="60"
         Stroke="Black"
         StrokeThickness="4"/>

    </Canvas>
```

Несмотря на то, что класс Line предоставляет свойство Fill, его установка не влияет на объект, так как Line не имеет внутренней области.

Другой общей фигурой является Ellipse. Создайте объект Ellipse, определив свойства Width и Height фигуры. Чтобы нарисовать круг, определите объект Ellipse значения которого Width и Height равны.



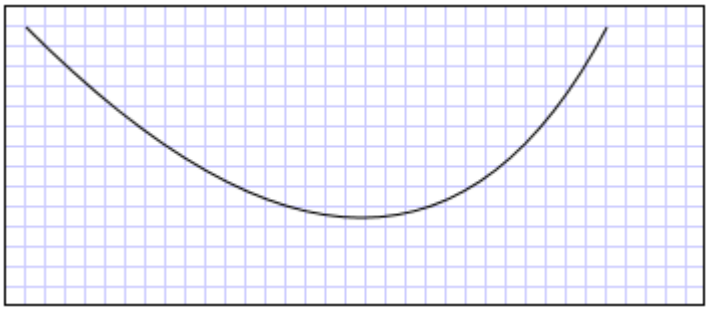
```
<Ellipse
Fill="Yellow"
Height="100"
Width="200"
StrokeThickness="2"
Stroke="Black"/>
```

Класс Path позволяет рисовать кривые линии и сложные фигуры. Эти кривые линии и фигуры описываются с помощью объектов Geometry. Чтобы использовать объект Path, создайте объект Geometry и используйте его, чтобы задать свойство Data объекта Path.

Имеется выбор из различных объектов Geometry. Классы LineGeometry, RectangleGeometry и EllipseGeometry описывают относительно простые фигуры. Для создания более сложных фигур или кривых линий используйте класс PathGeometry.

Объекты PathGeometry составлены из одного или нескольких объектов PathFigure; каждый объект PathFigure представляет отдельную "фигуру" или форму. Каждый объект PathFigure состоит из одного или нескольких объектов PathSegment, каждый из которых представляет переходную часть фигуры или формы. Типы сегментов включают следующие объекты: LineSegment, BezierSegment и ArcSegment.

В следующем примере объект Path используется для рисования кривой Безье второго порядка.



```
<Path Stroke="Black" StrokeThickness="1">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigureCollection>
          <PathFigure StartPoint="10,100">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <QuadraticBezierSegment Point1="200,200"
Point2="300,100" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>
        </PathFigureCollection>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

Все классы Line, Path, Polygon, Polyline и Rectangle имеют свойство Stretch. Это свойство определяет, как содержимое объекта Shape (фигура, которая будет нарисована) должно быть растянуто для заполнения пространства макета объекта Shape. Пространство макета объекта Shape — это количество пространства, которое система выделила для размещения объекта Shape или явным заданием параметра Width и Height, или с учетом параметров HorizontalAlignment и VerticalAlignment.

Свойство растягивания принимает одно из следующих значений:

None: содержимое объекта Shape не растягивается.

Fill: содержимое объекта Shape растягивается для заполнения пространства макета. Коэффициент пропорциональности не сохраняется.

Uniform: содержимое объекта Shape растягивается, насколько это возможно для заполнения пространства макета с сохранением исходного коэффициента пропорциональности.

UniformToFill: содержимое объекта Shape растягивается до полного заполнения пространства макета с сохранением исходного коэффициента пропорциональности.

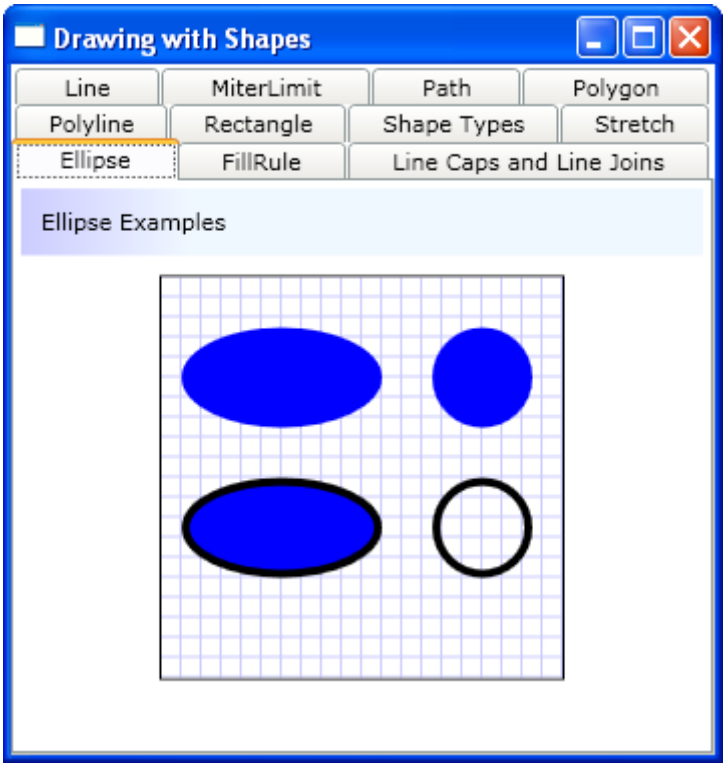
Обратите внимание, что когда содержимое объекта Shape растянуто, контур объекта Shape рисуется после растягивания.

7. Практический пример использования

В примере (Source\shapes) показано рисование фигур с помощью следующих элементов Shape: Ellipse, Line, Path, Polygon, Polyline и Rectangle.

В файлк app.xaml описано несколько общих ресурсов (кисти и стили). Главное окно приложения описано в SampleViewer.xaml. Оно содержит TabControl каждая страничка, которого отображает один из примеров в виде элементов Page описаних в остальных файлах

Вкладка Ellipse показывает различные режимы рисования эллипсов.



Вкладка Line Caps and Line Joins показывает пример различных режимов отрисовки концовок линий и соединений.

На остальных вкладках находятся примеры для разных фигур.

8. Трансформации

Transform определяет способ сопоставления или преобразования точек из одного координатного пространства в другое. Это сопоставление описано преобразованием Matrix, которое представляет собой коллекцию из трех строк с тремя столбцами, содержащую значения типа Double. Управляя значениями матрицы, можно поворачивать, масштабировать, наклонять и перемещать объект.

Не смотря на то, что WPF дает возможность непосредственно управлять значениями матрицы, она также предоставляет несколько классов Transform, позволяющих преобразовывать объект, не зная структуры матрицы. Например, класс ScaleTransform позволяет масштабировать объект с помощью определения его свойств ScaleX и ScaleY вместо управления матрицей преобразования. Аналогично класс RotateTransform позволяет повернуть объект, просто задав его свойство Angle.

WPF предоставляет следующие классы Transform для общих операций преобразования:

RotateTransform

Поворачивает элемент на значение, заданное в Angle.

ScaleTransform

Масштабирует элемент, используя коэффициенты, заданные в ScaleX и ScaleY.

SkewTransform

Наклоняет элемент на коэффициенты, заданные в AngleX и AngleY.

TranslateTransform

Сдвигает элемент на расстояние, заданное в X и Y.

Для совершения более сложных преобразований WPF предоставляет следующие два класса:

TransformGroup

Группирует несколько объектов TransformGroup в один объект Transform, к которому затем можно применять преобразования свойств.

MatrixTransform

Создает настраиваемые преобразования, не предоставленные другими классами Transform. При использовании MatrixTransform происходит непосредственное управление матрицей.

Одним из способов преобразования объекта является объявление соответствующего типа Transform и применение его к свойству преобразования объекта. Для различных типов объектов существуют различные типы свойств преобразования. далее перечислены несколько часто используемых типов WPF и их свойства преобразования.

Brush - Transform, RelativeTransform

ContainerVisual - Transform

DrawingGroup - Transform

FrameworkElement - RenderTransform, LayoutTransform

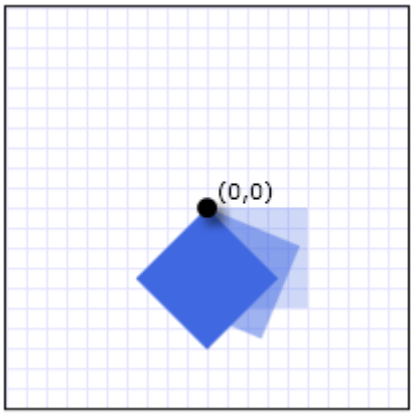
Geometry - Transform

TextEffect - Transform

UIElement – RenderTransform

При преобразовании объекта происходит не просто его преобразование, а преобразование пространства координат, в котором существует объект. По умолчанию преобразование центрируется в начале координат целевого объекта (0,0). Единственным исключением является TranslateTransform; TranslateTransform не имеет свойства центра, так как результат смещения не зависит от выбора центра смещения.

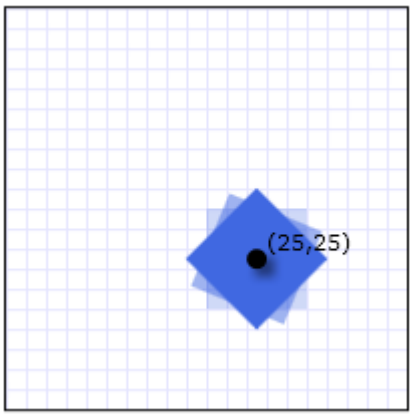
В следующем примере RotateTransform используется для поворота элемента Rectangle типа FrameworkElement на 45 градусов относительно его центра по умолчанию, (0, 0). На следующей иллюстрации приведен результат поворота.




```
<Canvas Width="200" Height="200">
  <Rectangle
    Canvas.Left="100" Canvas.Top="100"
    Width="50" Height="50"
    Fill="RoyalBlue" Opacity="1.0">
    <Rectangle.RenderTransform>
      <RotateTransform Angle="45" />
    </Rectangle.RenderTransform>
  </Rectangle>
</Canvas>
```

По умолчанию элемент поворачивается относительно своего левого верхнего угла, (0, 0). Классы RotateTransform, ScaleTransform и SkewTransform предоставляют свойства CenterX и CenterY, позволяющие задать точку, относительно которой применяется преобразование.

В следующем примере также используется RotateTransform для поворота элемента Rectangle на 45 градусов; однако, в этом случае свойства CenterX и CenterY заданы так, что центр RotateTransform (25, 25). На следующей иллюстрации приведен результат поворота.




```
<Canvas Width="200" Height="200">
  <Rectangle
    Canvas.Left="100" Canvas.Top="100"
    Width="50" Height="50"
    Fill="RoyalBlue" Opacity="1.0">
    <Rectangle.RenderTransform>
      <RotateTransform Angle="45" CenterX="25" CenterY="25" />
    </Rectangle.RenderTransform>
  </Rectangle>
</Canvas>
```

Чтобы применить преобразования к FrameworkElement, создайте Transform и примените его к одному из двух свойств, предоставленных классом FrameworkElement:

LayoutTransform – преобразование, применяемое перед проходом разметки. После применения преобразования система макета обрабатывает преобразованные размер и положение элемента.

RenderTransform – преобразование, которое изменяет положение элемента, но применяется после завершения прохода разметки. Используя свойства RenderTransform вместо свойства LayoutTransform, можно увеличить производительность системы.

Какое свойство следует использовать? Ввиду предоставляемого повышения производительности используйте свойство RenderTransform везде, где это возможно, особенно при использовании анимированных объектов Transform. При масштабировании, повороте или наклоне при-

меняется свойство LayoutTransform. Также требуется, чтобы родитель изменяемого элемента изменил свой размер в соответствии с изменениями своего дочернего элемента. Обратите внимание, что при использовании совместно со свойством LayoutTransform объекты TranslateTransform не вызывают видимых изменений в элементах. Это происходит потому, что, в ходе преобразования, система макета возвращает преобразуемый элемент на его исходную позицию.

В следующем примере для поворота кнопки по часовой стрелке на 45 градусов используется RotateTransform. Кнопка располагается в StackPanel, имеющей еще 2 кнопки.

По умолчанию RotateTransform поворачивается относительно точки (0, 0). Поскольку в примере не задано значение для центра, кнопка поворачивается относительно точки (0, 0), то есть верхнего левого угла. RotateTransform применяется к свойству RenderTransform.

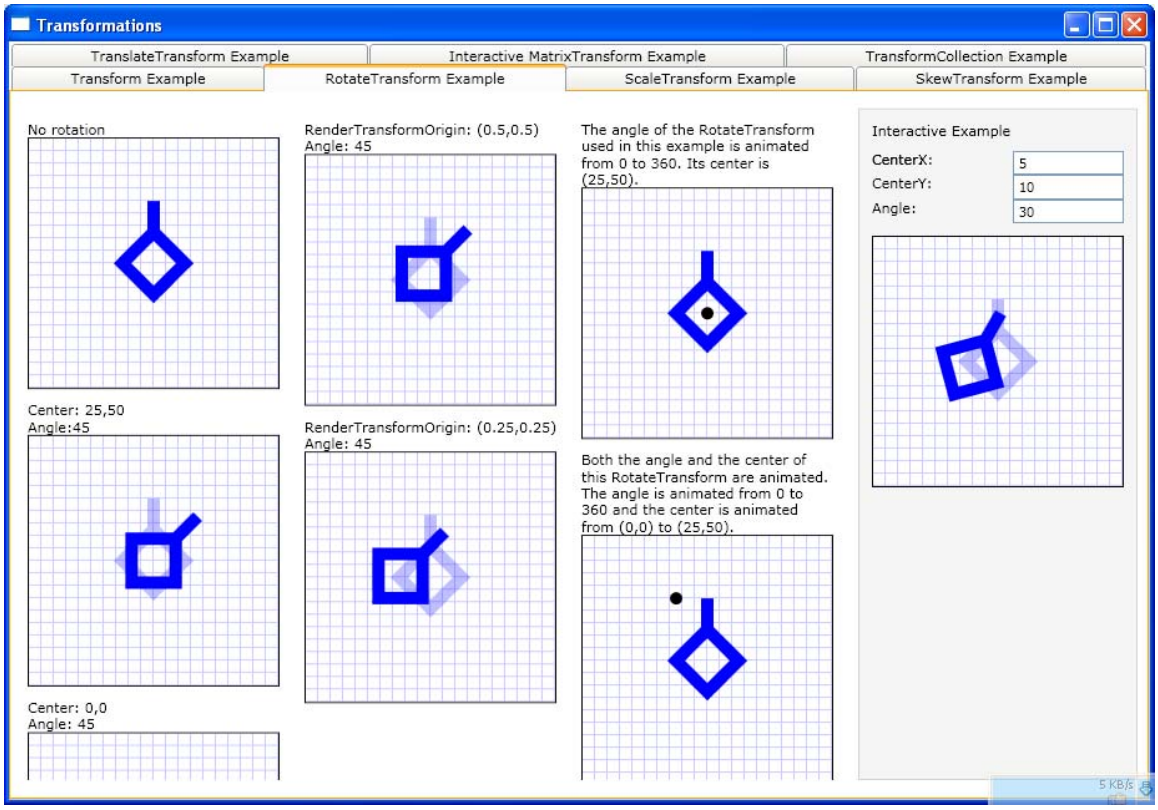
```
<Border Margin="30"
  HorizontalAlignment="Left" VerticalAlignment="Top"
  BorderBrush="Black" BorderThickness="1" >
  <StackPanel Orientation="Vertical">
    <Button Content="A Button" Opacity="1" />
    <Button Content="Rotated Button">
      <Button.RenderTransform>
        <RotateTransform Angle="45" />
      </Button.RenderTransform>
    </Button>
    <Button Content="A Button" Opacity="1" />
  </StackPanel>
</Border>
```

Поскольку происходит наследование от класса Animatable, классы Transform должны быть анимированными. Чтобы анимировать Transform, примените анимацию совместимого типа к требуемому свойству.

В следующем примере используется Storyboard и DoubleAnimation с RotateTransform, чтобы заставить Button вращаться при нажатии.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Button Animated RotateTransform Example"
  Background="White" Margin="50">
  <StackPanel>



    <Button Content="A Button"
      RenderTransformOrigin="0.5,0.5">
      <Button.RenderTransform>
        <RotateTransform x:Name="AnimatedRotateTransform" Angle="0" />
      </Button.RenderTransform>
      <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation
                Storyboard.TargetName="AnimatedRotateTransform"
                Storyboard.TargetProperty="Angle"
                To="360" Duration="0:0:1" FillBehavior="Stop" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Button.Triggers>
    </Button>

  </StackPanel>
</Page>
```

В примере (Source\trans) применение различных преобразований на практике.


Transformations
TranslateTransform Example
Interactive MatrixTransform Example
TransformCollection Example
Transform Example
RotateTransform Example
ScaleTransform Example
SkewTransform Example
No rotation
Center: 25,50
Angle:45
Center: 0,0
Angle: 45
RenderTransformOrigin: (0.5,0.5)
Angle: 45
RenderTransformOrigin: (0.25,0.25)
Angle: 45
The angle of the RotateTransform used in this example is animated from 0 to 360. Its center is (25,50).
Both the angle and the center of this RotateTransform are animated. The angle is animated from 0 to 360 and the center is animated from (0,0) to (25,50).
Interactive Example
CenterX:
5
CenterY:
10
Angle:
30

9. Отображение мультимедийной информации с помощью MediaElement

И MediaElement, и MediaPlayer используются для представления аудио, видео и видео с аудио. Оба типа могут управляться в интерактивном режиме или в режиме часов. Обоим типам требуется минимум Проигрыватель Windows Media 10 OCX для воспроизведения мультимедиа. Однако API-интерфейс рассчитаный на применение в различных сценариях.

MediaElement является UIElement, который принимается как содержимое многими элементами управления. Объекты мультимедиа, загруженные с помощью MediaPlayer, можно отобразить только при помощи VideoDrawing или при непосредственном взаимодействии с DrawingContext. MediaPlayer нельзя использовать в языке XAML.

Чтобы понять принцип воспроизведения мультимедиа в Windows Presentation Foundation (WPF), необходимо знание о различных режимах, в которых можно воспроизводить мультимедиа. И MediaElement, и MediaPlayer могут использоваться в двух различных режимах: независимом режиме и режиме часов. Режим мультимедиа определяется свойством Clock. Если Clock имеет значение null, то объект мультимедиа находится в независимом режиме. Если Clock имеет значение, отличное от NULL, то объект мультимедиа находится в режиме часов. По умолчанию объекты мультимедиа находятся в независимом режиме.

В независимом режиме содержимое мультимедиа управляет воспроизведением мультимедиа. Независимый режим позволяет следующее:

Класс Uri мультимедиа можно указывать непосредственно.

Воспроизведением мультимедиа можно управлять напрямую.

Свойства Position и SpeedRatio мультимедиа можно изменять.

Мультимедиа загружается присвоением значения свойству Source объекта MediaElement или путем вызова метода Open объекта MediaPlayer.

Для управления воспроизведением мультимедиа в независимом режиме можно использовать управляющие методы объекта мультимедиа. Доступные управляющие методы: Play, PauseClose и Stop. Для MediaElement интерактивное управление с использованием этих методов доступно только если LoadedBehavior имеет значение Manual. Эти методы недоступны, если объект мультимедиа находится в режиме часов.

В режиме часов MediaTimeline управляет воспроизведением мультимедиа. Режим часов имеет следующие характеристики:

Класс Uri мультимедиа задается неявно с помощью MediaTimeline.

Воспроизведение мультимедиа может управляться часами. Управляющие методы объекта мультимедиа использовать нельзя.

Объект мультимедиа загружается присвоением значения свойству Source объекта MediaTimeline, созданием часов из временной шкалы и назначением часов объекту мультимедиа. Объект мультимедиа также загружается таким образом, если MediaTimeline в Storyboard обращается в MediaElement.

Для управления воспроизведением мультимедиа в режиме часов должны использоваться управляющие методы класса ClockController. ClockController берется из свойства ClockController элемента MediaClock. При попытке использовать управляющие методы объектов MediaElement или MediaPlayer в режиме часов будет вызвано InvalidOperationException.

Добавить мультимедиа в приложение просто: нужно добавить элемент управления MediaElement в пользовательский интерфейс приложения и предоставить Uri объекту мультимедиа, который требуется включить. Все типы объектов мультимедиа, поддерживаемые программой Проигрыватель Windows Media 10, поддерживаются и в Windows Presentation Foundation . В следующем примере показано простое использование MediaElement в Язык XAML.

```
<!-- This page shows a simple usage of MediaElement -->
<Page x:Class="MediaElementExample.SimpleUsage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SimpleUsage"
        >
      <StackPanel Margin="20">
        <MediaElement Source="media/numbers-aud.wmv" />
      </StackPanel>
    </Page>
```

В этом примере объект мультимедиа воспроизводится автоматически по мере его загрузки. После завершения воспроизведения мультимедиа закрывается и все ресурсы, используемые мультимедиа, освобождаются (включая видеопамять). Это поведение по умолчанию для объекта MediaElement и оно управляется свойствами LoadedBehavior и UnloadedBehavior.

Свойства LoadedBehavior и UnloadedBehavior управляют поведением MediaElement, когда IsLoaded имеет значение true или false соответственно. Свойства MediaState устанавливаются, чтобы повлиять на поведение воспроизведения объектов мультимедиа. Например, LoadedBehavior по умолчанию имеет значение Play, а UnloadedBehavior по умолчанию имеет значение Close. Это означает, что после загрузки MediaElement и завершения предварительной пробы начинается воспроизведение мультимедиа. После завершения воспроизведения мультимедиа закрывается и освобождаются все ресурсы, используемые мультимедиа.

Свойства LoadedBehavior и UnloadedBehavior не являются единственным способом управления воспроизведением мультимедиа. В режиме часов часы могут управлять MediaElement, а интерактивные управляющие методы могут управлять, если LoadedBehavior имеет значение Manual. MediaElement обрабатывает эту конкуренцию управления, вычисляя следующие приоритеты:

UnloadedBehavior. В случае выгрузки мультимедиа. Гарантирует, что все ресурсы мультимедиа освобождаются по умолчанию, даже если MediaClock сопоставлен с MediaElement.

MediaClock. В случае, когда мультимедиа имеет Clock. Если мультимедиа выгружен, то MediaClock будет иметь эффект, пока UnloadedBehavior имеет значение Manual. Режим часов всегда переопределяет загруженное поведение MediaElement.

LoadedBehavior. В случае загрузки мультимедиа.

Интерактивные управляющие методы. В случае, если LoadedBehavior имеет значение Manual. Доступные управляющие методы: Play, PauseClose и Stop.

Для отображения MediaElement он должен иметь содержимое для отображения и значения его свойств ActualWidth и ActualHeight будут равными нулю до тех пор, пока не будет загружено содержимое. Для содержимого, содержащего только звук, эти свойства всегда равны нулю. Для видео после вызова события MediaOpened свойства ActualWidth и ActualHeight сообщат размер загруженного мультимедиа. Это означает, что до тех пор, пока мультимедиа не будет загружено, MediaElement не понадобится физического места в пользовательский интерфейс, если не задано свойство Width или Height.

Присвоение значения и свойству Width, и свойству Height приведет к растягиванию мультимедиа для заполнения области, предоставленной для MediaElement. Чтобы сохранить исходные пропорции мультимедиа, одно из свойств Width или Height должно быть задано, но не оба одновременно. Присвоение значения и свойству Width, и свойству Height приведет к отображению мультимедиа в элементе фиксированного размера, а это может быть нежелательным.

MediaElement полноправный компонент WPF и с ним можно творить все то же что и с другими. В следующем примере разметки элемент списка воспроизведения MediaElement повернут с помощью преобразования RotateTransform.

```
<MediaElement Source="media/numbers-aud.wmv">
  <MediaElement.LayoutTransform>
    <TransformGroup>
```

```
            <RotateTransform Angle="305" />
          </TransformGroup>
        </MediaElement.LayoutTransform>
      </MediaElement>
```

Экзаменационные задания

Написать программу для просмотра графических файлов.

Требования:

1) Вывод красивого логотипа при запуске программы. Можно с анимацией.
2) Формирование списка для просмотра добавлением нужных папок. Поиск в папках может проходить только на верхнем уровне или на всю глубину дерева.
3) Отображение всех картинок в виде уменьшенных значков.
4) Возможность полноэкранного просмотра (последовательно или случайно).
5) Слайдшоу с настройками эффектов.

Написать программу для игры в «Судоку».

Требования:

1) Вывод красивого логотипа при запуске программы. Можно с анимацией.
2) Сохранение и загрузка игры.
3) Различные уровни сложности.
4) Анимация и эффекты в игре.
5) Подсказки для игрока.