1. Поля ввода: TEXTBOXES

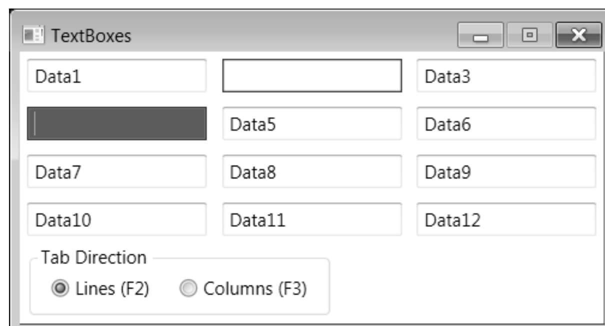

Рис. 1. Окно приложения TEXTBOXES

1.1. Дополнительное выделение активного поля ввода

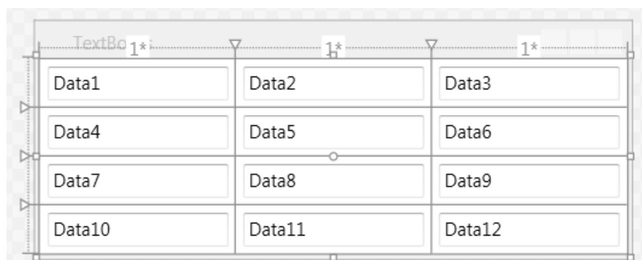

Рис. 2. Макет окна приложения TEXTBOXES (первый вариант)

При определении полей ввода в xaml-файле удобно вначале полностью задать первое поле, скопировать его в буфер обмена, а затем выполнять его вставку, корректируя имя поля, свойство Text и, при необходимости, значения свойств Grid.Row и Grid.Column (напомним, что в случае равенства 0 эти свойства можно не указывать).

```
<Window x:Class="TEXTBOXES.MainWindow"
    …
    Title="TextBoxes" SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanMinimize" >
  <Grid x:Name="grid1" >
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
```

```
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBox x:Name="textBox1" Grid.Row="0" Grid.Column="0"
        Margin="5" Text="Data1" MinWidth="120"/>
    <TextBox x:Name="textBox2" Grid.Row="0" Grid.Column="1"
        Margin="5" Text="Data2" MinWidth="120"/>
    <TextBox x:Name="textBox3" Grid.Row="0" Grid.Column="2"
        Margin="5" Text="Data3" MinWidth="120"/>
    <TextBox x:Name="textBox4" Grid.Row="1" Grid.Column="0"
        Margin="5" Text="Data4" MinWidth="120"/>
    <TextBox x:Name="textBox5" Grid.Row="1" Grid.Column="1"
        Margin="5" Text="Data5" MinWidth="120"/>
    <TextBox x:Name="textBox6" Grid.Row="1" Grid.Column="2"
        Margin="5" Text="Data6" MinWidth="120"/>
    <TextBox x:Name="textBox7" Grid.Row="2" Grid.Column="0"
        Margin="5" Text="Data7" MinWidth="120"/>
    <TextBox x:Name="textBox8" Grid.Row="2" Grid.Column="1"
        Margin="5" Text="Data8" MinWidth="120"/>
    <TextBox x:Name="textBox9" Grid.Row="2" Grid.Column="2"
        Margin="5" Text="Data9" MinWidth="120"/>
    <TextBox x:Name="textBox10" Grid.Row="3" Grid.Column="0"
        Margin="5" Text="Data10" MinWidth="120"/>
    <TextBox x:Name="textBox11" Grid.Row="3" Grid.Column="1"
        Margin="5" Text="Data11" MinWidth="120"/>
    <TextBox x:Name="textBox12" Grid.Row="3" Grid.Column="2"
        Margin="5" Text="Data12" MinWidth="120"/>
  </Grid>
</Window>
```

Для поля ввода textBox1 создайте обработчики событий GotFocus и LostFocus:

```
    <TextBox x:Name="textBox1" … GotFocus="textBox1_GotFocus"
        LostFocus="textBox1_LostFocus" />
```

После этого *переместите* полученные атрибуты в компонент Grid:

```
  <Grid x:Name="grid1" GotFocus="textBox1_GotFocus"
      LostFocus="textBox1_LostFocus" >
    …
    <TextBox x:Name="textBox1" … GotFocus="textBox1_GotFocus"
        LostFocus="textBox1_LostFocus" />
```

В файле MainWindow.xaml.cs в описание класса MainWindow добавьте поля

```
Brush backgr;
Brush foregr;
```

В конструктор класса MainWindow добавьте следующие операторы:

```
backgr = textBox1.Background;
foregr = textBox1.Foreground;
textBox1.Focus();
```

Наконец, определите в классе MainWindow ранее созданные обработчики:

```
private void textBox1_GotFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    tb.Foreground = Brushes.White;
    tb.Background = Brushes.Green;
}
private void textBox1_LostFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    tb.Foreground = foregr;
    tb.Background = backgr;
}
```

Результат. При получении фокуса любым полем ввода (т. е. при *активизации* поля ввода) изменяется его фон и цвет символов; при потере фокуса восстанавливается исходная цветовая настройка.

Недочет. При получении фокуса вертикальный курсор (*каретка*, caret), имеющий вид вертикальной линии, появляется в начале текста, тогда как удобнее, чтобы он располагался в его конце.

Исправление. Добавьте в метод textBox1_GotFocus оператор:

```
tb.Select(tb.Text.Length, 0);
```

Результат. Теперь при получении фокуса клавиатурный курсор располагается за последним символом текста.

1.2. Управление порядком обхода полей на форме

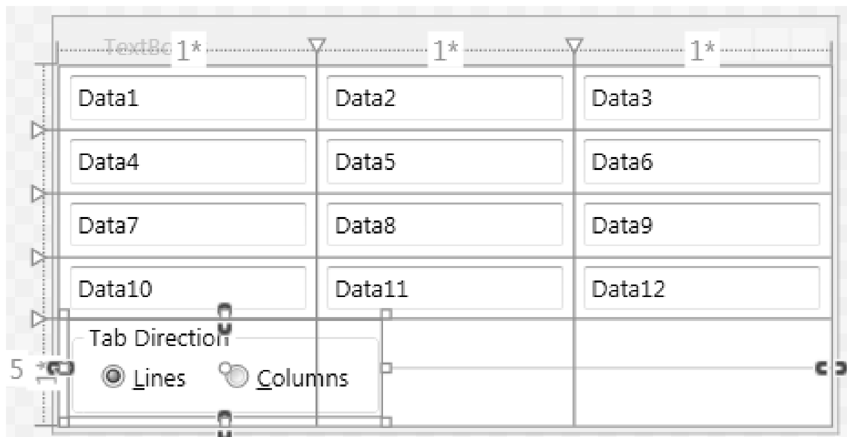

Рис. 3. Макет окна приложения TEXTBOXES (второй вариант)

```
<Window x:Class="TEXTBOXES.MainWindow"
        … >
  <Grid x:Name="grid1" GotFocus="textBox1_GotFocus"
      LostFocus="textBox1_LostFocus" >
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    …
    <TextBox x:Name="textBox12" Grid.Row="3" Grid.Column="2"
         Margin="5" Text="Data12" MinWidth="120"/>
    <GroupBox Grid.ColumnSpan="3" Grid.Row="4" Grid.Column="0"
        Header="Tab Direction" HorizontalAlignment="Left"
        Margin="5,0,5,5" VerticalAlignment="Center">
      <StackPanel Orientation="Horizontal">
        <RadioButton x:Name="radioButton1" Content="_Lines"
            Margin="10,5" IsChecked="True"
            Checked="radioButton1_Checked" />
        <RadioButton x:Name="radioButton2" Content="_Columns"
            Margin="10,5" Checked="radioButton2_Checked" />
      </StackPanel>
    </GroupBox>
  </Grid>
</Window>
```

```
private void radioButton1_Checked(object sender,
```

```
  RoutedEventArgs e)
{
    foreach (var e1 in grid1.Children)
    {
        TextBox tb = e1 as TextBox;
        if (tb != null)
            tb.TabIndex = int.MaxValue;
    }
}
private void radioButton2_Checked(object sender,
  RoutedEventArgs e)
{
    for (int i = 0; i < grid1.Children.Count - 1; i++)
    {
        TextBox tb = grid1.Children[i] as TextBox;
        tb.TabIndex = i * (int)Math.Pow(10, i % 3);
    }
}
```

Результат. Добавленные в окно радиокнопки предназначены для изменения порядка обхода полей ввода (теперь возможны два варианта обхода с помощью клавиш Tab и Shift+Tab: по строкам и по столбцам). Для переключения порядка обхода можно также использовать клавиатурные комбинации Alt+L и Alt+C.

Ошибка. При попытке выбрать радиокнопку (любым способом – с помощью щелчка мыши или путем нажатия клавиатурной комбинации) в программе возникает исключение. Это связано с тем, что для радиокнопок, как и для полей ввода, предусмотрены события GotFocus и LostFocus, при возникновении которых компонент Grid запускает обработчики textBox1_GotFocus и textBox1_LostFocus (эти обработчики запускаются для *любых* его дочерних компонентов, если для них предусмотрены данные события). При вызове обработчиков для радиокнопок свойство e.Source будет иметь тип RadioButton, который *нельзя* привести к типу TextBox. В такой ситуации операция as возвращает значение null, которое записывается в переменную tb, и при попытке обращения к любому свойству для «пустой» переменной tb возбуждается исключение NullReference-Exception.

Исправление. Дополните методы textBox1_GotFocus и textBox1_Lost-Focus:

```
private void textBox1_GotFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
```

```
    if (tb == null)
        return;
    tb.Foreground = Brushes.White;
    tb.Background = Brushes.Green;
    tb.Select(tb.Text.Length, 0);
}
private void textBox1_LostFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    if (tb == null)
        return;
    tb.Foreground = foregr;
    tb.Background = backgr;
}
```

Недочет. При любом из реализованных способов изменения порядка обхода полей текущее поле ввода теряет фокус (поскольку фокус принимает одна из радиокнопок).

Исправление. Определите для компонента grid1 обработчик события PreviewKeyDown и дополните текст радиокнопок:

```
<Grid x:Name="grid1" GotFocus="textBox1_GotFocus"
      LostFocus="textBox1_LostFocus"
      PreviewKeyDown="grid1_PreviewKeyDown" >
  …
      <RadioButton x:Name="radioButton1" Content="_Lines (F2)"
          Margin="10,5" IsChecked="True"
          Checked="radioButton1_Checked" />
      <RadioButton x:Name="radioButton2" Content="_Columns (F3)"
          Margin="10,5" Checked="radioButton2_Checked" />
```

```
private void grid1_PreviewKeyDown(object sender, KeyEventArgs e)
{
    switch (e.Key)
    {
        case Key.F2:
            radioButton1.IsChecked = true;
            break;
        case Key.F3:
            radioButton2.IsChecked = true;
            break;
    }
}
```

Результат. Теперь для настройки порядка обхода полей по строкам достаточно нажать клавишу F2, а по столбцам – F3, причем текущее поле ввода сохраняет фокус.

1.3. Проверка правильности введенных данных

В файле MainWindow.xaml.cs в описание класса MainWindow добавьте поля

```
Brush bordbr;
Thickness bordth;
```

Для поля ввода textBox1 создайте обработчик события TextChanged:

```
<TextBox x:Name="textBox1" …
    TextChanged="textBox1_TextChanged" />
```

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    if (tb.Text == "")
    {
        tb.BorderBrush = Brushes.Red;
        tb.BorderThickness = new Thickness(1.01);
    }
    else
    {
        tb.BorderBrush = bordbr;
        tb.BorderThickness = bordth;
    }
}
```

После создания обработчика textBox1_TextChanged *удалите* связанный с ним атрибут в xaml-файле:

```
<TextBox x:Name="textBox1" …
    TextChanged="textBox1_TextChanged" />
```

Наконец, дополните конструктор класса MainWindow следующим образом:

```
public MainWindow()
{
    InitializeComponent();
    grid1.AddHandler(TextBox.TextChangedEvent,
      new TextChangedEventHandler(textBox1_TextChanged));
    backgr = textBox1.Background;
    foregr = textBox1.Foreground;
    bordbr = textBox1.BorderBrush;
```

```
    bordth = textBox1.BorderThickness;
    textBox1.Focus();
}
```

Результат. Если активное поле ввода является пустым, то вокруг него рисуется красная рамка, которая сохраняется и при потере фокуса этим полем. Такой способ позволяет наглядно информировать пользователя о том, что введенное им значение является недопустимым.

Недочет. Причина, по которой пустое поле выделяется как ошибочное, может быть непонятна пользователю.

Исправление. Дополните текст метода textBox1_TextChanged:

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    if (tb.Text == "")
    {
        tb.BorderBrush = Brushes.Red;
        tb.BorderThickness = new Thickness(1.01);
        tb.ToolTip = "Поле не должно быть пустым";
    }
    else
    {
        tb.BorderBrush = bordbr;
        tb.BorderThickness = bordth;
        tb.ToolTip = null;
    }
}
```

Результат. Теперь при наведении курсора мыши на поле, обведенное красной рамкой (не обязательно активное), возникает *всплывающая подсказка* (tool tip): «Поле не должно быть пустым». Для заполненных полей подсказка не отображается.

1.4. Блокировка окна с ошибочными данными

Как правило, программы, в которые введены ошибочные данные, не запрещают пользователю перемещаться по различным полям ввода, но при этом блокируют выполнение действий, связанных с окончательной обработкой всего введенного набора данных (например, сохранение данных в файле или их пересылка по сети). Реализуем аналогичное поведение для нашего проекта.

```
<Window x:Class="TEXTBOXES.MainWindow"
    … Closing="Window_Closing" >
  …
```

```
</Window>
```

```
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    bool res = false;
    foreach (var e1 in grid1.Children)
        if ((e1 as Control).BorderBrush == Brushes.Red)
        {
            res = true;
            break;
        }
    e.Cancel = res;
}
```

Результат. Наличие пустого поля ввода не препятствует переходу в другие поля, однако пустое поле помечается как ошибочное. При наличии хотя бы одного ошибочного поля окно нельзя закрыть.

2. Обработка событий от мыши: MOUSE

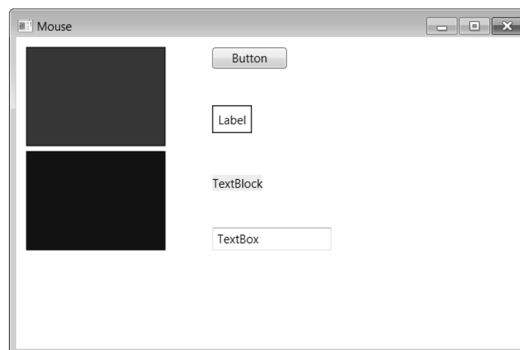

Рис. 4. Окно приложения MOUSE

2.1. Перетаскивание панели с помощью мыши

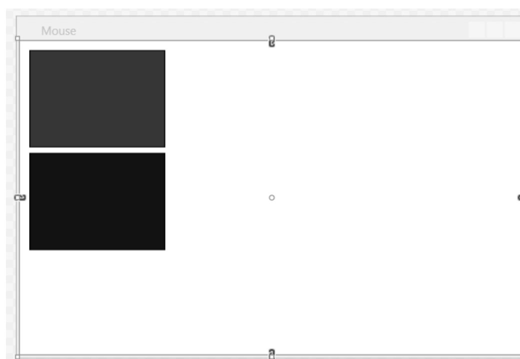

Рис. 5. Макет окна приложения MOUSE (первый вариант)

```
<Window x:Class="MOUSE.MainWindow"
    …
    Title="Mouse" Height="350" Width="525"
    WindowStartupLocation="CenterScreen" >
  <Canvas x:Name="canvas1" Background="White" >
    <Rectangle x:Name="rect1" Fill="Red" Height="100"
        Canvas.Left="10" Stroke="Black" Canvas.Top="10"
        Width="140" />
    <Rectangle x:Name="rect2" Fill="Blue" Height="100"
        Canvas.Left="10" Stroke="Black" Canvas.Top="115"
        Width="140" />
  </Canvas>
</Window>
```

В описание класса MainWindow добавьте поле

```
Point p;
```

Для компонента rect1 создайте обработчики событий MouseDown и MouseMove:

```
<Rectangle x:Name="rect1" … MouseDown="rect1_MouseDown"
    MouseMove="rect1_MouseMove" />
```

```
private void rect1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    if (e.Source == canvas1)
        return;
    var a = e.Source as Rectangle;
    p = e.GetPosition(a);
}
private void rect1_MouseMove(object sender, MouseEventArgs e)
{
    if (e.Source == canvas1)
    {
        Title = "Mouse";
        return;
    }
    var a = e.Source as Rectangle;
    Point q = e.GetPosition(a);
    Title = string.Format("Mouse - {0} {1}", a.Name, q);
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        Canvas.SetLeft(a, Canvas.GetLeft(a) + q.X - p.X);
        Canvas.SetTop(a, Canvas.GetTop(a) + q.Y - p.Y);
    }
}
```

После создания обработчиков *переместите* связанные с ними атрибуты MouseDown="rect1_MouseDown" и MouseMove ="rect1_MouseMove" в компонент Canvas:

```
<Canvas x:Name="canvas1" … MouseDown="rect1_MouseDown"
    MouseMove="rect1_MouseMove" >
  <Rectangle x:Name="rect1" … MouseDown="rect1_MouseDown"
      MouseMove="rect1_MouseMove" />
```

Результат. При перемещении курсора мыши над любым прямоугольником в заголовке окна выводится имя прямоугольника и текущие значения *локальных* координат мыши (относительно прямоугольника). Если при этом удерживается нажатой левая кнопка мыши, то прямоугольник

перемещается по окну (*перетаскивается* мышью). При перемещении курсора мыши на свободную часть окна восстанавливается исходный заголовок «Mouse».

Недочет. Прямоугольник rect1 при перетаскивании может «заслоняться» прямоугольником rect2. При этом если курсор мыши окажется над прямоугольником rect2, то курсор будет «захвачен» прямоугольником rect2 (поскольку этот прямоугольник располагается сверху и поэтому перехватывает события от мыши). Было бы удобнее всегда делать «верхним» тот компонент, который перетаскивается в данный момент.

Исправление. Добавьте в класс MainWindow новое поле

```
int maxz;
```

В метод rect1_MouseDown добавьте оператор:

```
Canvas.SetZIndex(a, ++maxz);
```

2.2. Изменение размеров компонента с помощью мыши. Захват мыши и его особенности

В описание класса MainWindow добавьте поле

```
Size s;
```

В метод rect1_MouseDown добавьте оператор

```
s = new Size(a.ActualWidth, a.ActualHeight);
```

В метод rect1_MouseMove добавьте фрагмент

```
else
if (e.RightButton == MouseButtonState.Pressed)
{
    a.Width = s.Width + q.X - p.X;
    a.Height = s.Height + q.Y - p.Y;
}
```

Результат. При перемещении мыши над любым прямоугольником с нажатой *правой* кнопкой происходит изменение *размеров* прямоугольника (левая кнопка по-прежнему используется для изменения положения прямоугольника в окне).

Недочет. Для того чтобы получить прямоугольник малого размера, необходимо «зацепить» его правой кнопкой мыши около правого нижнего угла, поскольку изменение размеров прекращается, как только курсор мыши покинет область прямоугольника.

Отмеченный недочет можно исправить, если использовать возможность *захвата мыши* (mouse capture). Следует заметить, что в библиотеке Windows Forms захват мыши выполняется автоматически, тогда как в библиотеке WPF им надо управлять явным образом.

Явление захвата состоит в том, что если нажать над компонентом какую-либо кнопку мыши, то этот компонент «захватит» мышь и «заставит» передавать ему все сообщения от мыши (даже если курсор мыши покинет

компонент) до тех пор, пока кнопка мыши не будет отпущена (причем это событие тоже будет обработано компонентом, ранее захватившим мышь). Используя захват мыши, мы получаем более полный контроль над действиями, связанными с перетаскиванием компонента и изменением его размеров, однако при этом могут возникнуть особые ситуации, для которых потребуется предусматривать специальную обработку.

Исправление. Добавьте новый оператор *в начало* метода rect1_MouseDown:

```
private void rect1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    (e.Source as FrameworkElement).CaptureMouse();
    if (e.Source == canvas1)
        return;
    var a = e.Source as Rectangle;
    p = e.GetPosition(a);
    Canvas.SetZIndex(a, ++maxz);
    s = new Size(a.ActualWidth, a.ActualHeight);
}
```

Для компонента rect1 создайте обработчик события MouseUp, после чего *переместите* связанный с ним атрибут в компонент Canvas:

```
<Canvas x:Name="canvas1" … MouseUp="rect1_MouseUp" >
  <Rectangle x:Name="rect1" … MouseUp="rect1_MouseUp" />
```

```
private void rect1_MouseUp(object sender, MouseButtonEventArgs e)
{
    (e.Source as FrameworkElement).ReleaseMouseCapture();
}
```

Результат. Теперь прямоугольник можно перетащить даже на область вне окна приложения: если при этом не отпускать кнопку мыши, то прямоугольник можно вернуть обратно на видимую часть окна. Аналогично, уменьшив размеры прямоугольника практически до нулевых (так, что курсор мыши покинет область прямоугольника, а одна или обе координаты, отображаемые в заголовке окна, станут отрицательными), можно затем восстановить их.

Эффект захвата может проявиться в нашей программе и по-другому: если нажать кнопку мыши на свободной части окна (т. е. на компоненте Canvas), а затем переместить курсор мыши на один из прямоугольников, то заголовок окна не изменится (он по-прежнему будет иметь вид «Mouse»). В данной ситуации мышь захватывается *компонентом Canvas* (поскольку для него тоже вызывается обработчик rect1_MouseDown), поэтому при перемещении мыши над прямоугольником событие MouseMove

передается не прямоугольнику, а захватившему мышь компоненту Canvas. Стоит в этой ситуации отпустить мышь над прямоугольником, как любое ее последующее перемещение будет перехвачено этим прямоугольником, что проявится в изменении заголовка окна.

Ошибка. Если попытаться сделать размеры прямоугольника меньше нулевых, то возникнет исключение, связанное с тем, что свойства Width и Height *не могут принимать отрицательные значения*. Кроме того, если перетащить прямоугольник целиком за левую или верхнюю границу окна и *отпустить кнопку мыши*, то доступ к прямоугольнику окажется невозможным.

Исправление. Измените завершающую часть метода rect1_MouseMove следующим образом:

```
if (e.LeftButton == MouseButtonState.Pressed)
{
    Canvas.SetLeft(a, Math.Max(0, Canvas.GetLeft(a) + q.X - p.X));
    Canvas.SetTop(a, Math.Max(0, Canvas.GetTop(a) + q.Y - p.Y));
}
else
if (e.RightButton == MouseButtonState.Pressed)
{
    a.Width = Math.Max(20, s.Width + q.X - p.X);
    a.Height = Math.Max(20, s.Height + q.Y - p.Y);
}
```

Результат. Теперь перетаскивать прямоугольник за пределы левой или верхней границы окна невозможно, и, кроме того, определен *минимальный* размер прямоугольника, равный 20 × 20 (напомним, что размеры в WPF задаются в *аппаратно-независимых единицах*, равных 1/96 дюйма).

2.3. Использование дополнительных курсоров

Добавьте в метод rect1_MouseDown следующие операторы:

```
if (e.ChangedButton == MouseButton.Left)
    a.Cursor = Cursors.Hand;
else
if (e.ChangedButton == MouseButton.Right)
    a.Cursor = Cursors.SizeNWSE;
```

Добавьте в метод rect1_MouseUp оператор:

```
(e.Source as FrameworkElement).Cursor = null;
```

Результат. В режиме изменения размеров курсор мыши принимает вид диагональной двунаправленной стрелки, а в режиме перетаскивания – «указывающей руки». После выхода из этих режимов восстанавливается стандартный вид курсора.

2.4. *Обработка ситуации с одновременным нажатием двух кнопок мыши*

Наша программа будет прекрасно работать до тех пор, пока пользователю не придет в голову мысль нажать левую кнопку мыши *при нажатой правой* (или наоборот). Для определенности опишем поведение программы в ситуации, когда при нажатой на прямоугольнике левой кнопке мыши была дополнительно нажата правая: в момент нажатия второй кнопки вид курсора изменится на диагональную стрелку, однако при последующем перемещении мыши по-прежнему будет выполняться перемещение окна (а не изменение его размеров). Если в этой ситуации отпустить левую кнопку мыши (оставив нажатой правую), то курсор примет стандартный вид, но при перемещении мыши будут *изменяться размеры* прямоугольника, причем при выходе курсора мыши за границу прямоугольника режим перемещения будет отменен (так как при ранее выполненном отпускании левой кнопки был отменен режим захвата мыши).

Чтобы избежать подобных нежелательных эффектов, следует более детально анализировать состояние кнопок мыши в обработчиках, связанных с их нажатием и отпусканием. При этом следует учитывать, что в обработчике события MouseMove мы *вначале* анализируем левую кнопку мыши, а *затем* правую. Таким образом, естественно считать, что левая кнопка *имеет более высокий приоритет*: если она нажата (даже одновременно с правой), то должно выполняться *перемещение* прямоугольника, а не изменение его размеров (и вид курсора должен учитывать эту особенность).

Измените методы rect1_MouseDown и rect1_MouseUp:

```
private void rect1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    (e.Source as UIElement).CaptureMouse();
    if (e.Source == canvas1)
        return;
    var a = e.Source as Rectangle;
    p = e.GetPosition(a);
    Canvas.SetZIndex(a, ++maxz);
    s = new Size(a.ActualWidth, a.ActualHeight);
    if (e.LeftButton == MouseButtonState.Pressed)
        a.Cursor = Cursors.Hand;
    else
    if (e.RightButton == MouseButtonState.Pressed)
        a.Cursor = Cursors.SizeNWSE;
}
```

```
private void rect1_MouseUp(object sender, MouseButtonEventArgs e)
{
    var a = e.Source as FrameworkElement;
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        if (a != canvas1)
            a.Cursor = Cursors.Hand;
    }
    else
    if (e.RightButton == MouseButtonState.Pressed)
    {
        if (a != canvas1)
            a.Cursor = Cursors.SizeNWSE;
    }
    else
    {
        (e.Source as FrameworkElement).ReleaseMouseCapture();
        (e.Source as FrameworkElement).Cursor = null;
    }
}
```

Результат. Внесенные исправления полностью исключают описанное выше аномальное поведение. При нажатии и отпускании любых кнопок мыши в любом порядке на компоненте Canvas курсор не изменяется. При нажатии и отпускании любых кнопок мыши в любом порядке на прямоугольниках вид курсора всегда соответствует правильному режиму: это режим перетаскивания, если нажата левая кнопка (даже если одновременно нажата правая), или режим изменения размера, если нажата правая кнопка, а левая отпущена. В любом случае захват мыши отменяется только при отпускании *всех* ее кнопок. Отметим также, что наша программа корректно реагирует и на действия со *средней* кнопкой мыши (т. е. с колесиком, которое также можно нажимать): нажатие и отпускание этой кнопки никак не влияет на текущий режим программы.

2.5. Перетаскивание компонентов любого типа

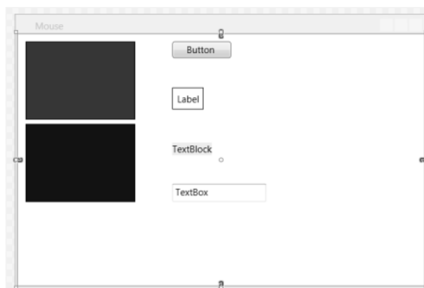

Рис. 6. Макет окна приложения MOUSE (второй вариант)

Добавьте к компоненту Canvas новые компоненты различного типа:

```
<Canvas x:Name="canvas1" … >
  …
  <Button x:Name="button1" Content="Button" Canvas.Left="197"
      Canvas.Top="10" Width="75"/>
  <Label x:Name="label1" BorderBrush="Black" BorderThickness="1"
      Content="Label" Canvas.Left="197" Canvas.Top="69"/>
  <TextBlock x:Name="textBlock1" Background="Yellow"
      Canvas.Left="197" Text="TextBlock" Canvas.Top="139"/>
  <TextBox x:Name="textBox1" Height="23" Canvas.Left="197"
      Text="TextBox" Canvas.Top="192" Width="120"/>
</Canvas>
```

В каждый из методов rect1_MouseDown и rect1_MouseMove внесите следующее изменение:

```
var a = e.Source as Rectangle;
var a = e.Source as FrameworkElement;
```

Результат. Добавленные компоненты (кнопка, два вида меток и поле ввода) обрабатываются *почти* так же, как и прямоугольники: их размер и положение можно изменять с помощью мыши (особенности, связанные с кнопкой и полем ввода, будут рассмотрены далее).

Недочет. Кнопку и поле ввода нельзя перемещать при нажатой левой кнопке мыши, подобно другим компонентам. Это связано с тем, что для данных компонентов нажатие левой кнопки мыши обрабатывается особым образом, а действия, определенные в обработчиках событий MouseDown, MouseMove и MouseUp, игнорируются. Тем не менее для этих компонентов тоже можно обеспечить выполнение требуемых действий, если вместо событий MouseDown, MouseMove и MouseUp обрабатывать *предшествующие им* события PreviewMouseDown, PreviewMouseMove и PreviewMouseUp.

Исправление. Измените три атрибута для компонента Canvas в xaml-файле (к имени каждого атрибута надо добавить префикс Preview):

```
<Canvas x:Name="canvas1" Background="White"
    PreviewMouseDown="rect1_MouseDown"
    PreviewMouseMove="rect1_MouseMove"
    PreviewMouseUp="rect1_MouseUp">
```

Результат. Теперь режим перемещения и изменения размеров работает одинаково для всех компонентов окна, причем для кнопки и поля ввода это не препятствует выполнению *дополнительных действий*, определенных для данных компонентов. Например, при нажатии на кнопку Button левой кнопкой мыши она переходит в «нажатое» состояние, а если установить поле ввода TextBox у левой границы окна и продолжить сдвигать

курсор мыши влево так, что курсор будет перемещаться по тексту, содержащемуся в поле ввода, то произойдет *выделение* части текста.

Указанные дополнительные действия определены в обработчиках для событий мыши, которые «встроены» в компоненты Button и TextBox. Несмотря на то что эти обработчики встроены в код компонентов, *их можно отключить*; для этого достаточно добавить в каждый из методов rect1_MouseDown и rect1_MouseMove следующий оператор (метод rect1_MouseUp изменять не требуется):

```
e.Handled = true;
```

Теперь при нажатии левой кнопкой мыши на кнопке Button она не будет переходить в нажатое состояние.

3. Перетаскивание (Drag & Drop): ZOO

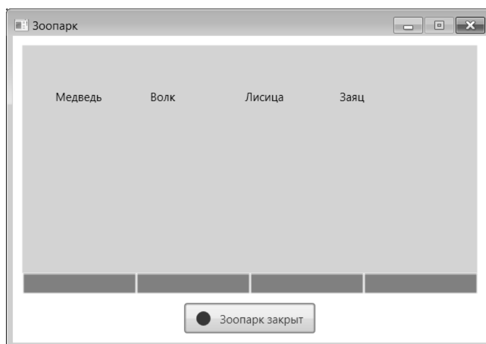

Рис. 7. Окно приложения ZOO

3.1. Перетаскивание меток в окне

Рис. 8. Макет окна приложения ZOO (первый вариант)

```
<Window x:Class="ZOO.MainWindow"
    …
    Title="Зоопарк" SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanMinimize" >
  <StackPanel >
    <Canvas x:Name="canvas1" Margin="10" Width="480" Height="240"
        Background="LightGray" AllowDrop="True"
        Drop="canvas1_Drop" >
      <TextBlock x:Name="label1" Canvas.Left="30" Text="Медведь"
          Canvas.Top="41" Padding="5" />
      <TextBlock x:Name="label2" Canvas.Left="130" Text="Волк"
          Canvas.Top="41" Padding="5" />
      <TextBlock x:Name="label3" Canvas.Left="230" Text="Лисица"
          Canvas.Top="41" Padding="5" />
      <TextBlock x:Name="label4" Canvas.Left="330" Text="Заяц"
          Canvas.Top="41" Padding="5" />
```

```
    </Canvas>
  </StackPanel>
</Window>
```

```
private void canvas1_Drop(object sender, DragEventArgs e)
{
    if (!(e.Source is Canvas))
        return;
    TextBlock src = e.Data.GetData(typeof(TextBlock))
      as TextBlock;
    Point p = e.GetPosition(canvas1);
    Canvas.SetLeft(src, p.X - src.ActualWidth / 2);
    Canvas.SetTop(src, p.Y - src.ActualHeight / 2);
}
```

Для компонента label1 создайте обработчик события MouseDown, после чего переместите соответствующий атрибут в компонент Canvas:

```
<Canvas x:Name="canvas1" … MouseDown="label1_MouseDown" >
  <TextBlock x:Name="label1" … MouseDown="label1_MouseDown" />
```

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    TextBlock t = e.Source as TextBlock;
    if (t == null)
        return;
    if (e.ChangedButton == MouseButton.Left)
        DragDrop.DoDragDrop(t, t, DragDropEffects.Move);
}
```

Результат. Метки с названиями зверей можно перетаскивать с помощью левой кнопки мыши по компоненту Canvas. В процессе перетаскивания метки-«источника» (source) она остается на месте, однако вид курсора мыши изменяется, что является признаком режима перетаскивания. В качестве «приемника» (target) пока определен лишь компонент Canvas: при отпускании над ним кнопки мыши происходит перемещение метки зверя на указанную позицию.

В дальнейшем в текстах обработчиков, связанных с перетаскиванием, будем использовать имена src и trg для объектов-*источников* и объектов-*приемников* соответственно.

Недочет 1. Если перетащить одну метку на другую и отпустить кнопку мыши, то ничего не произойдет. В подобной ситуации (при перемещении источника над недопустимым приемником) желательно, чтобы курсор имел вид запрещающего знака.

Исправление. Для компонента canvas1 создайте обработчик события DragEnter и укажите *этот же обработчик* для события DragOver:

```
<Canvas x:Name="canvas1" … DragEnter="canvas1_DragEnter"
    DragOver="canvas1_DragEnter" >
```

```
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Source is Canvas)
        return;
    e.Handled = true;
    e.Effects = DragDropEffects.None;
}
```

Результат. Теперь при перетаскивании одной метки на другую курсор мыши принимает вид запрещающего знака.

Недочет 2. Теперь в начальный момент перетаскивания курсор принимает вид запрещающего знака. Это нежелательно, так как может ввести в заблуждение пользователя, который решит, что он сделал что-то не так.

Исправление. Откорректируйте последний оператор в методе canvas1_DragEnter:

```
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Source is Canvas)
        return;
    e.Handled = true;
    e.Effects = e.Data.GetData(typeof(TextBlock)) == e.Source ?
        DragDropEffects.Move : DragDropEffects.None;
}
```

Результат. В начальный момент перетаскивания курсор мыши остается разрешающим. Перетаскивание одной метки на другую по-прежнему запрещено.

3.2. Перетаскивание меток в поля ввода

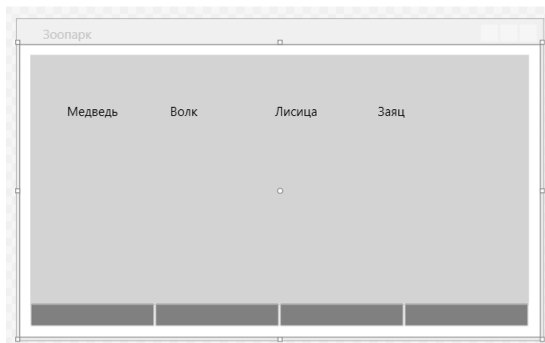

Рис. 9. Макет окна приложения ZOO (второй вариант)

```
<Window x:Class="ZOO.MainWindow"
    … >
  <StackPanel >
    <Canvas x:Name="canvas1" Margin="10,10,10,0" … >
      …
    </Canvas>
    <UniformGrid x:Name="grid1" Margin="10,0,10,10" Columns="4"
        AllowDrop="True">
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
    </UniformGrid>
  </StackPanel>
</Window>
```

Для компонента grid1 создайте обработчики событий PreviewDragEnter и Drop и задайте для события PreviewDragOver обработчик, уже созданный для события PreviewDragEnter:

```
<UniformGrid x:Name="grid1" …
    PreviewDragEnter="grid1_PreviewDragEnter"
    PreviewDragOver="grid1_PreviewDragEnter" Drop="grid1_Drop" >
```

```
private void grid1_PreviewDragEnter(object sender,
  DragEventArgs e)
{
    var trg = e.Source as TextBox;
    if (trg == null)
        return;
    e.Handled = true;
    e.Effects = trg.Text == "" ?
        DragDropEffects.Move : DragDropEffects.None;
}
private void grid1_Drop(object sender, DragEventArgs e)
{
    var trg = e.Source as TextBox;
    if (trg == null)
        return;
    var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
    trg.Text = src.Text;
    src.Visibility = Visibility.Hidden;
}
```

Результат. Теперь приемником может также служить любое *незаполненное* поле ввода («пустая клетка»). При перетаскивании метки на незаполненное поле ввода «зверь попадает в клетку» (текст метки отображается в поле ввода). Перетаскивание метки на уже заполненное поле ввода пока запрещено, хотя в следующем пункте это действие станет доступным.

3.3. Взаимодействие меток при их перетаскивании друг на друга

```
<Window x:Class="ZOO.MainWindow"
    … >
  <StackPanel >
    <Canvas … >
      <TextBlock x:Name="label1" … Tag="4" />
      <TextBlock x:Name="label2" … Tag="3" />
      <TextBlock x:Name="label3" … Tag="2" />
      <TextBlock x:Name="label4" … Tag="1" />
    </Canvas>
    <UniformGrid … >
      <TextBox … Tag="0" />
      <TextBox … Tag="0" />
      <TextBox … Tag="0" />
      <TextBox … Tag="0" />
    </UniformGrid>
  </StackPanel>
</Window>
```

Измените имеющиеся обработчики canvas1_DragEnter, canvas1_Drop и grid1_Drop следующим образом:

```
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Source is Canvas)
        return;
    e.Handled = true;
    e.Effects = e.Data.GetData(typeof(TextBlock)) == e.Source ?
      DragDropEffects.Move : DragDropEffects.None;
    e.Effects = DragDropEffects.Move;
}
private void canvas1_Drop(object sender, DragEventArgs e)
{
    if (!(e.Source is Canvas))
        return;
    if (e.Source is Canvas)
```

```
    {
        TextBlock src = e.Data.GetData(typeof(TextBlock))
          as TextBlock;
        Point p = e.GetPosition(canvas1);
        Canvas.SetLeft(src, p.X - src.ActualWidth / 2);
        Canvas.SetTop(src, p.Y - src.ActualHeight / 2);
    }
    else
    {
        var trg = e.Source as TextBlock;
        var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
        if ((src.Tag as string)[0] > (trg.Tag as string)[0])
        {
            Canvas.SetLeft(src, Canvas.GetLeft(trg));
            Canvas.SetTop(src, Canvas.GetTop(trg));
            trg.Visibility = Visibility.Hidden;
        }
        else
            src.Visibility = Visibility.Hidden;
    }
}
private void grid1_Drop(object sender, DragEventArgs e)
{
    var trg = e.Source as TextBox;
    if (trg == null)
        return;
    var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
    if ((src.Tag as string)[0] >= (trg.Tag as string)[0])
    {
        trg.Text = src.Text;
        trg.Tag = src.Tag;
    }
    src.Visibility = Visibility.Hidden;
}
```

Измените атрибуты PreviewDragEnter и PreviewDragOver для компонента grid1 в xaml-файле:

```
<UniformGrid x:Name="grid1" …
    PreviewDragEnter="canvas1_DragEnter"
    PreviewDragOver="canvas1_DragEnter" Drop="grid1_Drop">
```

После сделанных изменений обработчик grid1_PreviewDragEnter уже не будет связан ни с одним событием, поэтому его описание в классе MainWindow можно удалить.

Результат. При перетаскивании одного зверя на другого более сильный «поедает» более слабого. То же самое происходит, если один из зверей перетаскивается в клетку, уже занятую другим зверем. Заметим, что теперь события DragEnter и DragOver *всех* компонентов (и канвы, и меток, и полей ввода) связаны с одним и тем же обработчиком – canvas1_DragEnter.

Ошибка. Если при перетаскивании метки отпустить ее над ней самой, то метка исчезнет. Таким образом, *зверь поедает самого себя*.

Исправление. В методе canvas1_Drop *перед* оператором

```
if ((src.Tag as string)[0] > (trg.Tag as string)[0])
```

вставьте следующий фрагмент:

```
if (src == trg)
    return;
```

3.4. Действия в случае перетаскивания на недопустимый приемник

Измените метод label1_MouseDown:

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    var t = e.Source as TextBlock;
    if (t == null)
        return;
    if (e.ChangedButton == MouseButton.Left)
        if (DragDrop.DoDragDrop(t, t, DragDropEffects.All) ==
          DragDropEffects.None)
            t.Visibility = Visibility.Hidden;
}
```

Результат. Если перетаскивание метки-«зверя» завершается за пределами окна (в этом случае курсор перетаскивания имеет вид запрещающего знака), то зверь «убегает» из зоопарка, и его метка в окне исчезает.

3.5. Дополнительное выделение источника и приемника в ходе перетаскивания

Измените методы label1_MouseDown и canvas1_DragEnter:

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
```

```
    var t = e.Source as TextBlock;
    if (t == null)
        return;
    t.Foreground = Brushes.Red;
    if (e.ChangedButton == MouseButton.Left)
        if (DragDrop.DoDragDrop(t, t, DragDropEffects.All) ==
          DragDropEffects.None)
            t.Visibility = Visibility.Hidden;
    t.Foreground = Brushes.Black;
}
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    e.Handled = true;
    e.Effects = DragDropEffects.Move;
    var trg = e.Source as TextBlock;
    if (trg == null)
        return;
    trg.Background = Brushes.Yellow;
}
```

В методе canvas1_Drop после оператора

```
var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
```

добавьте новый оператор:

```
trg.Background = null;
```

Для компонента canvas1 создайте обработчик события DragLeave:

```
<Canvas x:Name="canvas1" … DragLeave="canvas1_DragLeave" >
```

```
private void canvas1_DragLeave(object sender, DragEventArgs e)
{
    e.Handled = true;
    var trg = e.Source as TextBlock;
    if (trg == null)
        return;
    trg.Background = null;
}
```

Результат. В режиме перетаскивания цвет текста метки-источника изменяется на красный, а текущая метка-приемник изображается на желтом фоне.

3.6. Настройка вида курсора в режиме перетаскивания

Для окна MainWindow создайте обработчик события GiveFeedback:

```
<Window x:Class="ZOO.MainWindow"
```

```
    … GiveFeedback="Window_GiveFeedback" >
```

```
private void Window_GiveFeedback(object sender,
  GiveFeedbackEventArgs e)
{
    if (e.Effects == DragDropEffects.Move)
    {
        e.UseDefaultCursors = false;
        Mouse.SetCursor(Cursors.Hand);
    }
    else
        e.UseDefaultCursors = true;
    e.Handled = true;
}
```

Результат. В режиме перетаскивания курсор мыши имеет вид «указывающей руки» (при выходе курсора за пределы окна он по-прежнему принимает вид запрещающего знака).

3.7. Информация о текущем состоянии программы. Кнопки с комбинированным содержимым

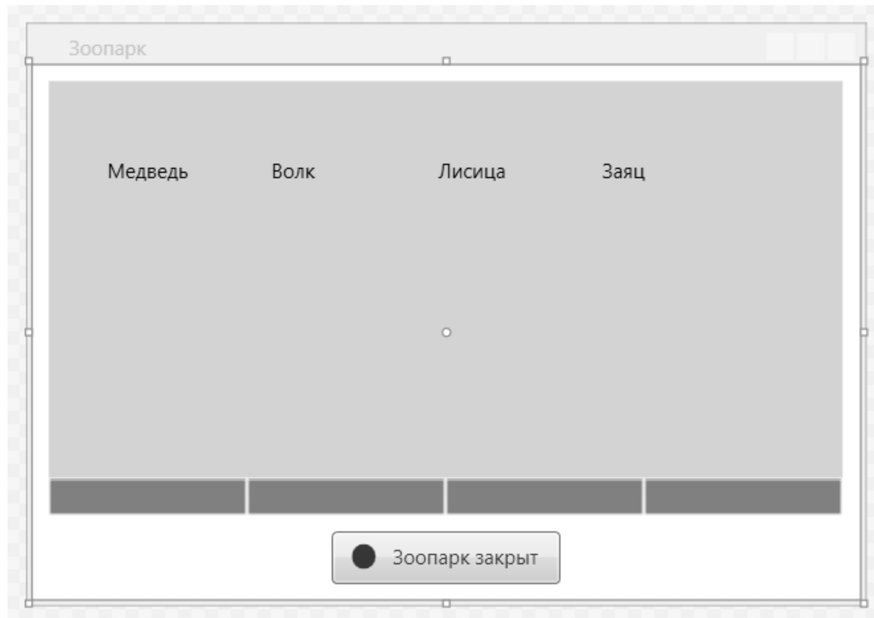

Рис. 10. Макет окна приложения ZOO (третий вариант)

```
<Window x:Class="ZOO.MainWindow"
    … >
  <StackPanel >
    <Canvas … >
      …
    </Canvas>
```

```
    <UniformGrid … >
      …
    </UniformGrid>
    <Button x:Name="button1" HorizontalAlignment="Center"
        Margin="0,0,0,10" >
      <StackPanel Margin="5,0" Orientation="Horizontal">
        <Ellipse x:Name="mark1" Margin="5" Fill="Red"
            Width="15" Height="15" />
        <TextBlock x:Name="caption1" Margin="5" Foreground="Red"
            Text="Зоопарк закрыт" />
      </StackPanel>
    </Button>
  </StackPanel>
</Window>
```

В метод label1_MouseDown добавьте операторы:

```
string s = "";
for (int i = 0; i < 4; i++)
{
    if (canvas1.Children[i].IsVisible)
        return;
    s += (grid1.Children[i] as TextBox).Text;
}
if (s == "")
    return;
mark1.Fill = Brushes.Green;
caption1.Foreground = Brushes.Green;
caption1.Text = "Зоопарк открыт";
```

Результат. В начале работы программы кнопка содержит изображение красного круга и текст «Зоопарк закрыт» (тоже красного цвета). Если в результате перетаскивания меток все они исчезли и при этом хотя бы одно поле ввода оказалось заполненным, то на кнопке выводится текст «Зоопарк открыт», а цвет оформления кнопки меняется с красного на зеленый. При попытке перетаскивания меток на кнопку или свободную часть окна слева или справа от кнопки курсор принимает вид запрещающего знака.

3.8. Восстановление исходного состояния

В описание класса MainWindow добавьте поля:

```
double[] startPosX = new double[4];
double startPosY;
```

В конструктор класса MainWindow добавьте операторы:

```
startPosY = Canvas.GetTop(canvas1.Children[0]);
```

```
for (int i = 0; i < 4; i++)
    startPosX[i] = Canvas.GetLeft(canvas1.Children[i]);
button1.Focus();
```

Для компонента button1 определите обработчик события Click:

```
<Button x:Name="button1" … Click="button1_Click" >
```

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    for (int i = 0; i < 4; i++)
    {
        var t = canvas1.Children[i];
        t.Visibility = Visibility.Visible;
        Canvas.SetTop(t, startPosY);
        Canvas.SetLeft(t, startPosX[i]);
        var tb = grid1.Children[i] as TextBox;
        tb.Text = "";
        tb.Tag = "0";
    }
    mark1.Fill = Brushes.Red;
    caption1.Foreground = Brushes.Red;
    caption1.Text = "Зоопарк закрыт";
}
```

Результат. Исходное положение меток-«зверей» сохраняется в полях startPosX и startPosY. В дальнейшем при нажатии на кнопку button1 исходное положение «зверей» восстанавливается, а «клетки» освобождаются. Кроме того, при запуске программы кнопка button1 становится активной (принимает фокус), что позволяет для ее нажатия просто нажать клавишу пробела или клавишу Enter.