1. Курсоры и иконки: CURSORS

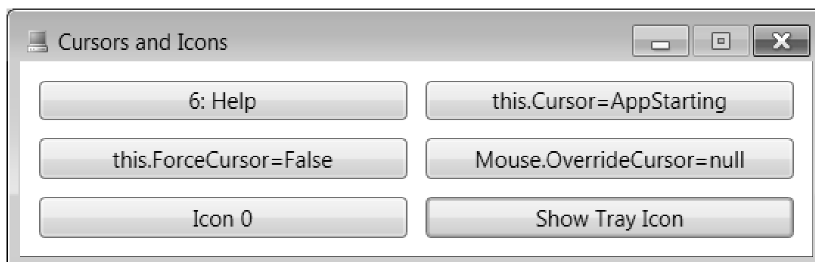


Рис. 1. Окно приложения CURSORS

1.1. Использование стандартных курсоров

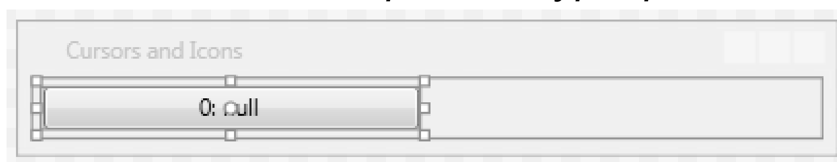


Рис. 2. Макет окна приложения CURSORS (первый вариант)

```
<Window x:Class="CURSORS.MainWindow"
    …
    Title="Cursors and Icons"
    WindowStartupLocation="CenterScreen"
    SizeToContent="WidthAndHeight" ResizeMode="CanMinimize" >
  <UniformGrid Margin="5" Columns="2" >
    <Button x:Name="button1" Content="0: null" MinWidth="200"
        Margin="5" PreviewMouseDown="button1_PreviewMouseDown" />
  </UniformGrid>
</Window>
```

В описание класса MainWindow добавьте вспомогательное поле:

```
List<Cursor> cur = new List<Cursor>(29);
```

В конструктор класса MainWindow добавьте следующий фрагмент:

```
cur.Add(null);
foreach (System.Reflection.PropertyInfo pi in
  typeof(Cursors).GetProperties())
    cur.Add((Cursor)pi.GetValue(null, null));
button1.Tag = 0;
```

И определите обработчик события PreviewMouseDown для компонента button1, уже указанный в xaml-файле:

```
private void button1_PreviewMouseDown(object sender,
  MouseButtonEventArgs e)
```

```
{
    int k = (int)button1.Tag, c = cur.Count;
    switch (e.ChangedButton)
    {
        case MouseButton.Left:
            k = (k + 1) % c;
            break;
        case MouseButton.Right:
            k = (k - 1 + c) % c;
            break;
    }
    button1.Content = k + ": " +
        (cur[k] == null ? "null" : cur[k].ToString());
    button1.Cursor = cur[k];
    button1.Tag = k;
    e.Handled = true;
}
```

Результат. Щелчок левой или правой кнопкой мыши на кнопке button1 приводит к смене вида курсора для данной кнопки (на кнопке при этом выводится порядковый номер курсора и его название). Перебор всех 28 стандартных курсоров выполняется циклически; порядок перебора курсоров соответствует порядку их размещения в выпадающем списке для свойства Cursor в окне Properties. При нажатии не левой, а правой кнопки мыши курсоры перебираются в обратном порядке. Предусмотрено также особое значение null для курсора кнопки, при котором используется курсор ее родительского компонента (это значение имеет номер 0).

1.2. Установка курсора для окна и приложения в целом

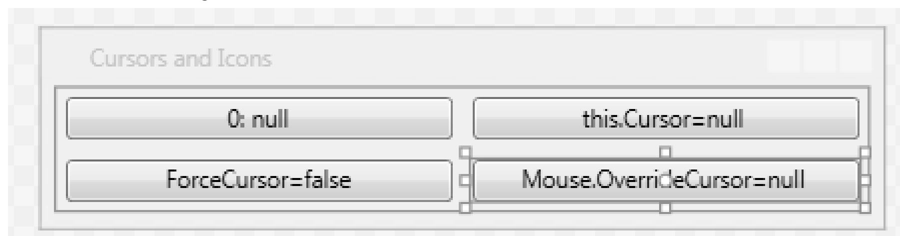


Рис. 3. Макет окна приложения CURSORS (второй вариант)

```
<Window x:Class="CURSORS.MainWindow"
    … >
  <UniformGrid Margin="5" Columns="2">
    <Button x:Name="button1" Content="0: null" MinWidth="200"
        Margin="5" PreviewMouseDown="button1_PreviewMouseDown"/>
    <Button x:Name="button2" Content="this.Cursor=null"
```

```
        MinWidth="200" Margin="5" Click="button2_Click"/>
    <Button x:Name="button3" Content="this.ForceCursor=false"
        MinWidth="200" Margin="5" Click="button3_Click"/>
    <Button x:Name="button4" Content="Mouse.OverrideCursor=null"
        MinWidth="200" Margin="5" Click="button4_Click"/>
  </UniformGrid>
</Window>
```

Определите обработчики события Click для компонентов button2, button3, button4, уже указанные в xaml-файле:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    Cursor = button1.Cursor;
    button2.Content = "this.Cursor=" +
        (Cursor == null ? "null" : Cursor.ToString());
}
private void button3_Click(object sender, RoutedEventArgs e)
{
    ForceCursor = !ForceCursor;
    button3.Content = "this.ForceCursor=" + ForceCursor;
}
private void button4_Click(object sender, RoutedEventArgs e)
{
    Mouse.OverrideCursor = Mouse.OverrideCursor == null ?
      button1.Cursor : null;
    button4.Content = "Mouse.OverrideCursor=" +
      (Mouse.OverrideCursor == null ? "null" :
      Mouse.OverrideCursor.ToString());
}
```

Результат. Нажатие на кнопку button2 распространяет действие нового курсора на все окно, включая те находящиеся в окне кнопки, для которых свойство Cursor равно null. Нажатие на кнопку button3 устанавливает для *всех* компонентов окна курсор окна, независимо от того, задан или нет для компонента какой-либо курсор (для этого используется свойство окна ForceCursor, которое полагается равным true). Нажатие на кнопку button4 изменяет свойство Mouse.OverrideCursor, которое *устанавливает курсор для всего приложения*, независимо от настроек любых других курсоров (действие свойства Mouse.OverrideCursor отменяется, если положить его равным null).

1.3. Использование в программе дополнительных курсоров

Кроме стандартных курсоров в программе можно использовать дополнительные курсоры. Файлы с курсорами имеют расширение .cur; их можно найти, например, в подкаталоге Cursors каталога Windows. Заметим, что в приложениях WPF (в отличие от приложений Windows Forms) можно использовать цветные курсоры, а также анимированные курсоры (файлы с расширением .ani). Иконка любого файла, содержащего курсор, соответствует изображению этого курсора, что позволяет быстро ознакомиться с содержимым имеющихся файлов курсоров, используя Проводник.

Выберите два cur-файла и скопируйте их в каталог проекта CURSORS под именами C1.cur и C2.cur.

Загрузите в среду Visual Studio проект CURSORS, если это еще не сделано, и выполните команду меню Project | Add Existing Item… (Shift+Alt+A). В появившемся диалоговом окне выберите файл C1.cur (имя файла можно ввести непосредственно в строке ввода; можно также указать в выпадающем списке «Files of type» вариант «All Files» и выбрать файл C1.cur из списка всех файлов, содержащихся в каталоге). Нажмите клавишу Enter или кнопку Add. В результате файл C1.cur будет добавлен к проекту CURSORS и появится в окне Solution Explorer. Аналогичными действиями добавьте к проекту файл C2.cur.

Если выделить один из добавленных файлов в окне Solution Explorer, то в окне Properties появится список его свойств. Проверьте, что свойство Build Action для этих файлов равно значению Resource.

Файлы, добавленные в приложение описанными выше действиями, являются *встроенными ресурсами приложения*. Эти файлы встраиваются непосредственно в исполняемый файл приложения, причем к ним можно обращаться по именам как в xaml-файле, так и в программном коде.

В данном приложении мы будем использовать ресурсы в программном коде (в проектах TEXTEDIT версии 4 и COLORS ресурсы будут указываться в xaml-файле).

Добавьте в конструктор класса MainWindow следующий фрагмент:

```
var rs = Application.GetResourceStream(new
  Uri("pack://application:,,,/C" + i + ".cur"));
cur.Add(new Cursor(rs.Stream));
```

Результат. При создании окна новые курсоры загружаются в программу и добавляются к списку доступных курсоров. Заметим, что для курсоров, импортированных из файлов, метод ToString возвращает строку None.

1.4. Работа с иконками

Небольшие изображения, называемые *иконками*, *значками* или *пикто-*

граммами, хранятся в файлах с расширением .ico. В версии Visual Studio 2005, 2008, 2010 включались графические библиотеки (в виде архивов с именами – VS2005ImageLibrary.zip, VS2008ImageLibrary.zip, VS2010ImageLibrary.zip), содержащие большой набор ico-файлов, однако в последних версиях подобные библиотеки отсутствуют. Впрочем, в Интернете имеется множество сайтов, с которых можно скачать коллекции иконок, да и найти несколько ico-файлов на компьютере не составляет труда.

Добавление к проекту ico-файлов выполняется аналогично добавлению cur-файлов. Для определенности будем считать, что к проекту добавлены файлы Computer.ico и Folder.ico. Как и в случае cur-файлов, после добавления ico-файла в проект следует проверить, что его свойство Build Action равно значению Resource.

```
<Window x:Class="CURSORS.MainWindow"
    … >
  <UniformGrid Margin="5" Columns="2">
    …
    <Button x:Name="button5" Content="Icon 0" MinWidth="200"
        Margin="5" Click="button5_Click"/>
  </UniformGrid>
</Window>
```

В описание класса MainWindow добавьте поле

```
BitmapImage[] ico = new BitmapImage[2];
```

В конструктор класса MainWindow добавьте следующий фрагмент:

```
ico[0] = new BitmapImage(new
  Uri("pack://application:,,,/Computer.ico"));
ico[1] = new BitmapImage(new
  Uri("pack://application:,,,/Folder.ico"));
Icon = ico[0];
button5.Tag = 0;
```

И определите обработчик Click кнопки button5, уже указанный в xaml-файле:

```
private void button5_Click(object sender, RoutedEventArgs e)
{
    int k = ((int)button5.Tag + 1) % 2;
    button5.Content = "Icon " + k;
    button5.Tag = k;
    Icon = ico[k];
}
```

Результат. Кнопка button5 изменяет иконку приложения как в заголовке окна, так и на его кнопке, находящейся на панели задач.

1.5. Размещение иконки в области уведомлений. Использование объектов из библиотеки Windows Forms

На последнем этапе разработки проекта CURSORS мы добавим к нему возможность отображения его иконки в *области уведомлений* (notification area, traybar) панели задач.

В библиотеке Windows Forms для этих целей был предусмотрен невизуальный компонент NotifyIcon. В библиотеке WPF аналогичное средство отсутствует. Тем не менее мы можем реализовать данную возможность, использовав в нашем проекте класс NotifyIcon из библиотеки Windows Forms.

Прежде всего, надо подключить необходимые компоненты библиотеки Windows Forms. Для этого щелкните правой кнопкой мыши на пункте References в окне Solution Explorer, выберите из появившегося меню команду Add Reference…, отметьте в появившемся списке стандартных компонентов библиотеки .NET пункты System.Windows.Forms и System.Drawing (около этих пунктов должны появиться «галочки») и закройте окно, нажав кнопку OK. После этого выбранные компоненты появятся в разделе References.

Указывать новые пространства имен в директивах using не следует, так как это приведет к многочисленным конфликтам, связанным с совпадением имен классов из библиотек Windows Forms и WPF. Поэтому для классов из библиотеки Windows Forms придется указывать их полные имена.

В приложение необходимо также добавить файл с иконкой, которую мы хотим связать с объектом NotifyIcon (уже имеющиеся иконки-ресурсы использовать не удастся, так как для ресурсов, применяемых в библиотеке Windows Forms, надо указать другое значение свойства Build Action). Предположим, что новый файл с иконкой имеет имя ComputerWF.ico. Его необходимо загрузить как встроенный ресурс и *изменить для него значение свойства Build Action на Embedded Resource*.

Теперь можно приступать к модификации xaml- и cs-файлов.

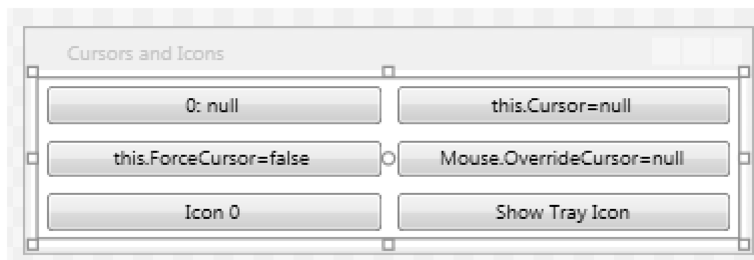


Рис. 4. Макет окна приложения CURSORS (третий вариант)

```
<Window x:Class="CURSORS.MainWindow"
    … >
  <UniformGrid Margin="5" Columns="2">
```

```
  …
  <Button x:Name="button6" Content="Show Tray Icon"
      MinWidth="200" Margin="5" Click="button6_Click"/>
 </UniformGrid>
</Window>
```

В описание класса MainWindow добавьте поле

```
System.Windows.Forms.NotifyIcon notifyIcon1 =
  new System.Windows.Forms.NotifyIcon();
```

В конструктор класса MainWindow добавьте следующий фрагмент, в котором настраиваются свойства объекта notifyIcon1:

```
notifyIcon1.Icon = new System.Drawing.Icon(GetType(),
  "ComputerWF.ico");
notifyIcon1.Visible = false;
notifyIcon1.Text = "Icon in Traybar";
notifyIcon1.Click += notifyIcon1_Click;
```

И определите два обработчика: для события Click кнопки button6, который уже указан в xaml-файле (и для которого, следовательно, уже имеется заготовка) и для события Click объекта notifyIcon1 (этот объект мы создали программно, поэтому текст его обработчика придется вводить полностью):

```
private void button6_Click(object sender, RoutedEventArgs e)
{
    bool b = (string)button6.Content == "Show Tray Icon";
    notifyIcon1.Visible = b;
    ShowInTaskbar = !b;
    button6.Content = b ? "Hide Tray Icon" : "Show Tray Icon";
}
private void notifyIcon1_Click(object sender, EventArgs e)
{
    button6_Click(null, null);
}
```

Результат. Нажатие на кнопку button6 скрывает кнопку приложения на панели задач и отображает связанную с ним иконку в области уведомлений в правой части панели задач. При наведении курсора на эту иконку возникает всплывающая подсказка «Icon in Traybar». Повторное нажатие кнопки button6 (или щелчок на иконке, расположенной в области уведомлений) восстанавливает исходное представление кнопки приложения на панели задач.

2. Табличное приложение с заставкой: TRIGFUNC

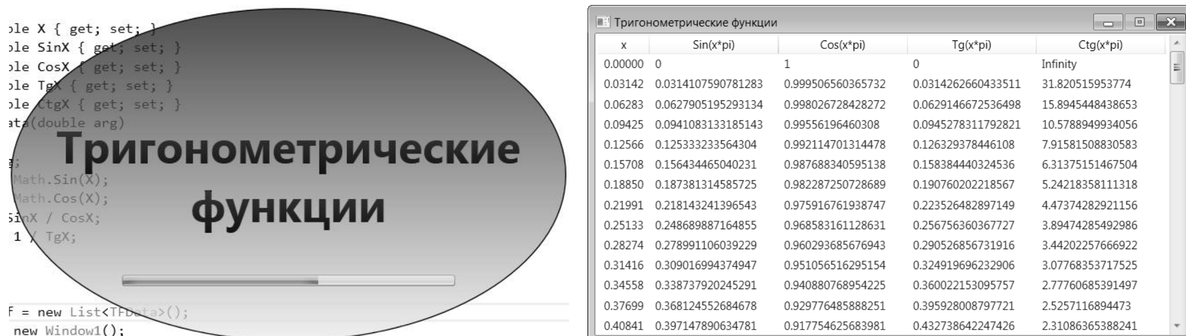


Рис. 5. Окна приложения TRIGFUNC

2.1. Формирование таблицы значений тригонометрических функций

Рис. 6. Макет окна MainWindow приложения TRIGFUNC

```
<Window x:Class="TRIGFUNC.MainWindow"
    …
    Title="Тригонометрические функции" Height="350"
    SizeToContent="Width" ResizeMode="CanMinimize"
    WindowStartupLocation="CenterScreen"
    Loaded="Window_Loaded" >
  <ListView x:Name="listView1" >
    <ListView.View >
      <GridView x:Name="gridView1" >
        <GridViewColumn Header="x"
            DisplayMemberBinding="{Binding Path=X,
              StringFormat='{}{0:f5}'}" />
```

```
        <GridViewColumn Header="Sin(x*pi)"
            DisplayMemberBinding="{Binding Path=SinX}"/>
        <GridViewColumn Header="Cos(x*pi)"
            DisplayMemberBinding="{Binding Path=CosX}"/>
        <GridViewColumn Header="Tg(x*pi)"
            DisplayMemberBinding="{Binding Path=TgX}"/>
        <GridViewColumn Header="Ctg(x*pi)"
            DisplayMemberBinding="{Binding Path=CtgX}"/>
      </GridView>
    </ListView.View>
  </ListView>
</Window>
```

В класс MainWindow добавьте описание вспомогательного вложенного класса TFData и поля tf, содержащего список объектов класса TFData:

```
class TFData
{
    public double X { get; set; }
    public double SinX { get; set; }
    public double CosX { get; set; }
    public double TgX { get; set; }
    public double CtgX { get; set; }
    public TFData(double arg)
    {
        X = arg;
        SinX = Math.Sin(X);
        CosX = Math.Cos(X);
        TgX = SinX / CosX;
        CtgX = 1 / TgX;
    }
}
List<TFData> tf = new List<TFData>();
```

Для окна MainWindow определите обработчик события Loaded, указанный в xaml-файле:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    int n = 7, nMax = 1000001;
    string[] args = Environment.GetCommandLineArgs();
    if (args.Length > 1)
        try
        {
```

```
            int n0 = int.Parse(args[1]);
            if (n0 < 2 || n0 > nMax)
                throw new Exception();
            else
                n = n0;
        }
        catch
        {
            string s = string.Format("Неверный параметр: {0}\n" +
              "Допустимые значения: от 2 до {1}", args[1], nMax);
            MessageBox.Show(s, "Ошибка", MessageBoxButton.OK,
              MessageBoxImage.Error);
            Close();
            return;
        }
    double step = 1.0 / (n - 1);
    for (int i = 0; i < n; i++)
        tf.Add(new TFData(Math.PI * i * step));
    listView1.ItemsSource = tf;
}
```

Результат. Программа заполняет компонент-таблицу listView1 значениями тригонометрических функций с аргументами от 0 до π радиан на сетке из n равноотстоящих точек. Число точек n можно указать в качестве параметра командной строки; допускаются значения от 2 до 1000001. При запуске программы из среды Visual Studio параметры можно указать в поле «Command line arguments» группы настроек Debug в окне свойств проекта (напомним, что данное окно вызывается командой «Project | <*Имя проекта*> Properties»). Если параметр указан неверно, то выводится сообщение об ошибке и программа немедленно завершает работу. Если параметр не указан, то количество точек полагается равным 7.

Хотя размеры окна изменять нельзя, допустимо менять ширину отдельных столбцов, зацепив мышью за разделительную линию между их заголовками и переместив ее в нужном направлении (при этом ширина окна автоматически изменится; это обеспечивается благодаря настройке окна SizeToContent="Width"). Более того, зацепив мышью за заголовок некоторого столбца, можно перетащить его на новую позицию, поменяв тем самым порядок следования столбцов.

Недочет 1. Поскольку для столбцов мы не указали значения Width, их ширина определяется по содержимому, однако только по тому содержимому, которое *отображается в окне в начале программы*. Если ниже (в невидимой части) присутствуют более длинные значения, то это никак

не будет учитываться. В качестве примера на рис. 69 приводится завершающая часть таблицы, построенной по 10000 точкам, в которой ширина трех последних столбцов недостаточна для отображения всех цифр в полученных числах.

Тригонометрические функции

x	Sin(x*pi)	Cos(x*pi)	Tg(x*pi)	Ctg(x*pi)
3.13782	0.0037702792806485	-0.99999289247181	-0.00377030607820522	-265.23045589869
3.13814	0.0034560906484161	-0.99999402770088	-0.0034561128934651	-289.34253450764
3.13845	0.00314190167501345	-0.99999506421475	-0.00314191718284193	-318.27700789219
3.13876	0.00282771239145506	-0.99999600201332	-0.00282772369665672	-353.64134097766
3.13908	0.00251352282875726	-0.99999684109650	-0.00251353076875839	-397.84673115180
3.13939	0.00219933301793508	-0.99999758146421	-0.00219933833711356	-454.68220288126
3.13971	0.00188514299000402	-0.99999822311637	-0.00188514633968968	-530.46279694371
3.14002	0.00157095277597959	-0.99999876605292	-0.00157095471445456	-636.55558673898
3.14034	0.00125676240687687	-0.99999921027381	-0.00125676339937584	-795.69471906696
3.14065	0.000942571913712315	-0.99999955577899	-0.00094257233242274	-1060.9265364597
3.14096	0.000628381327500585	-0.99999980256843	-0.00062838145156291	-1591.3900665157
3.14128	0.000314190679258144	-0.99999995064210	-0.00031419069476593	-3182.7804472216
3.14159	1.22460635382238E-16	-1	-1.22460635382238E-1	-8.1658893641919

Рис. 7. Вид окна приложения TRIGFUNC с недостаточной шириной столбцов

Разумеется, указанный недочет не является серьезным, так как пользователь может самостоятельно изменять ширину столбцов. Однако всетаки опишем способ его исправления, поскольку он может оказаться полезным в аналогичных ситуациях.

Исправление. Определите обработчик события LayoutUpdated для окна:

```
<Window x:Class="TRIGFUNC.MainWindow"
    …
    LayoutUpdated="Window_LayoutUpdated" >
```

```
private void Window_LayoutUpdated(object sender, EventArgs e)
{
    double maxWidth = gridView1.Columns
      .Select(e1 => e1.ActualWidth).Max() + 10;
    for (int i = 1; i < gridView1.Columns.Count; i++)
        gridView1.Columns[i].Width = maxWidth;
    LayoutUpdated -= Window_LayoutUpdated;
}
```

Результат. После определения реальных размеров каждого столбца (при этом, как было отмечено выше, учитываются только те данные, которые сразу будут отображаться на экране) ширина столбцов со второго по последний *пересчитывается еще раз*: она полагается равной максимальному из ранее определенных значений ширины *плюс* 10 аппаратно-

независимых единиц. Этого достаточно, чтобы все данные полностью отображались в таблице (рис. 70). Кроме того, все столбцы со значениями тригонометрических функций теперь имеют одинаковую ширину. Указанное действие выполняется только после *первого* определения реальных размеров компонентов окна. Заметим, что центрирование окна выполняется правильно, т. е. с учетом увеличившейся ширины окна.

Тригонометрические функции

x	Sin(x*pi)	Cos(x*pi)	Tg(x*pi)	Ctg(x*pi)
3.13782	0.0037702792806485	-0.999992892471814	-0.00377030607820522	-265.230455898697
3.13814	0.0034560906484161	-0.999994027700881	-0.00345611128934651	-289.342534507644
3.13845	0.00314190167501345	-0.999995064214751	-0.00314191718284193	-318.277007892193
3.13876	0.00282771239145506	-0.999996002013324	-0.00282772369665672	-353.641340977664
3.13908	0.00251352282875726	-0.999996841096505	-0.00251353076875839	-397.846731151801
3.13939	0.00219933301793508	-0.999997581464213	-0.00219933833711356	-454.68220288126
3.13971	0.00188514299000402	-0.999998223116375	-0.00188514633968968	-530.462796943717
3.14002	0.00157095277597959	-0.999998766052926	-0.00157095471445456	-636.555586738987
3.14034	0.00125676240687687	-0.999999210273814	-0.00125676339937584	-795.694719066963
3.14065	0.000942571913712315	-0.999999555778995	-0.000942572332422743	-1060.9265364597
3.14096	0.000628381327500585	-0.999999802568434	-0.000628381451562919	-1591.39006651579
3.14128	0.000314190679258144	-0.999999950642107	-0.000314190694765934	-3182.78044722165
3.14159	1.22460635382238E-16	-1	-1.22460635382238E-16	-8.16588936419192E+15

Рис. 8. Вид окна приложения TRIGFUNC после корректировки ширины столбцов

2.2. Отображение окна-заставки при загрузке программы

Добавьте к проекту новое окно Window1, выполнив действия, описанные в начале раздела, посвященного проекту WINDOWS (п. 2.1).

Рис. 9. Макет окна Window1 приложения TRIGFUNC (первый вариант)

Window1.xaml

```
<Window x:Class="TRIGFUNC.Window1"
    …
    Title="Window1" Height="300" AllowsTransparency="True"
    Width="500" WindowStartupLocation="CenterScreen"
```

```
    WindowStyle="None" Background="Transparent"
    ResizeMode="NoResize" ShowInTaskbar="False" >
  <Grid x:Name="grid1" >
    <Ellipse Stroke="Black" >
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <GradientStop Color="#BF0000FF" Offset="0"/>
          <GradientStop Color="#BFFFFF00" Offset="1"/>
        </LinearGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <TextBlock TextWrapping="Wrap"
        Text="Тригонометрические функции" FontSize="40"
        TextAlignment="Center" VerticalAlignment="Center"
        FontWeight="Bold" >
      <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <GradientStop Color="Red" Offset="0"/>
          <GradientStop Color="Blue" Offset="1"/>
        </LinearGradientBrush>
      </TextBlock.Foreground>
    </TextBlock>
  </Grid>
</Window>
```

Файл Window1.xaml.cs изменять не требуется.

В класс MainWindow добавьте поле

```
Window1 win1 = new Window1();
```

В *начало* метода Window_Loaded добавьте операторы:

```
LayoutUpdated -= Window_LayoutUpdated;
Mouse.OverrideCursor = Cursors.AppStarting;
win1.Owner = this;
win1.Show();
```

В *конец* метода Window_Loaded добавьте операторы:

```
win1.Hide();
Mouse.OverrideCursor = null;
```

Результат. В течение загрузки главного окна и заполнения табличного списка listView1 на экране отображается окно Window1 – *заставка*, свидетельствующая о том, что программа запущена и в данный момент выполняет инициализирующие действия. Заставка имеет форму эллипса, не содержит заголовка и является полупрозрачной; ее нельзя перемещать по экрану. Для фона заставки используется вертикальная градиентная за-

ливка (от синего цвета к желтому); такой же вариант заливки, только с другими цветами (от красного к синему), использован для текста заставки. Курсор мыши на заставке принимает вид курсора ожидания окончания загрузки программы. При отображении главного окна заставка исчезает.

Недочет 1. Требуемые вычисления обычно выполняются так быстро, что рассмотреть заставку не удается из-за краткого времени ее отображения (некоторое замедление в ходе начальных вычислений возникает только для числа точек, близкого к максимальному).

Исправление. В методе Window_Loaded *перед* оператором

```
win1.Hide();
```

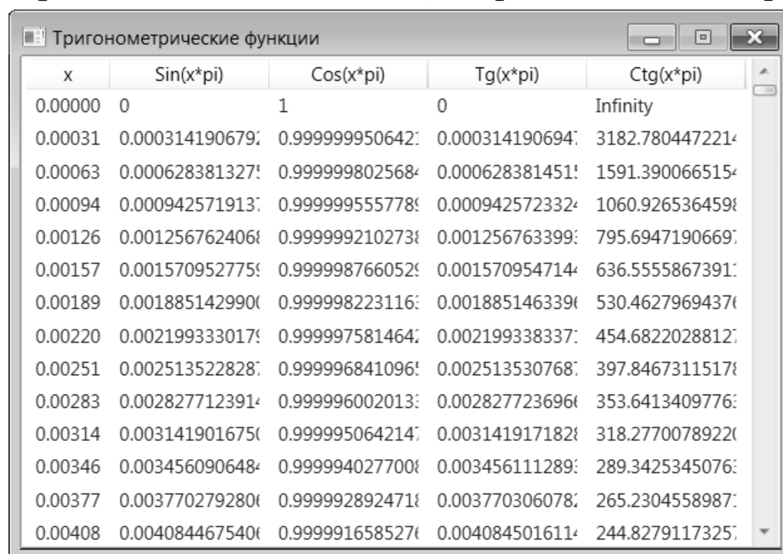
вставьте оператор

```
System.Threading.Thread.Sleep(1000);
```

Результат. После завершения инициализирующих действий, но перед скрытием окна-заставки программа приостанавливает работу на 1 секунду.

Недочет 2. Второй недочет оказывается неожиданным: ширина столбцов в главном окне стала пересчитываться неправильно!

Причем, судя по виду окна, в котором даже начальные данные отображаются не полностью (рис. 72), это означает, что обработчик Window_LayoutUpdated сработал еще *до того*, как были определены требуемые размеры столбцов. Эксперименты с программой показывают, что, действительно, из-за вызова win1.Show() возникает дополнительное (первое по счету) событие LayoutUpdated для главного окна, при котором фактические размеры столбцов таблицы *еще определяются неверно*.



x	Sin(x*pi)	Cos(x*pi)	Tg(x*pi)	Ctg(x*pi)
0.00000	0	1	0	Infinity
0.00031	0.000314190679	0.999999950642	0.000314190694	3182.780447221
0.00063	0.000628381327	0.999999802568	0.000628381451	1591.390066515
0.00094	0.000942571913	0.999999555778	0.000942572332	1060.926536459
0.00126	0.001256762406	0.999999210273	0.001256763399	795.6947190669
0.00157	0.001570952775	0.999998766052	0.001570954714	636.5555867391
0.00189	0.001885142990	0.999998223116	0.001885146339	530.4627969437
0.00220	0.002199333017	0.999997581464	0.002199338337	454.6822028812
0.00251	0.002513522828	0.999996841096	0.002513530768	397.8467311517
0.00283	0.002827712391	0.999996002013	0.002827723696	353.6413409776
0.00314	0.003141901675	0.999995064214	0.003141917182	318.2770078922
0.00346	0.003456090648	0.999994027700	0.003456111289	289.3425345076
0.00377	0.003770279280	0.999992892471	0.003770306078	265.2304558987
0.00408	0.004084467540	0.999991658527	0.004084501611	244.8279117325

Рис. 10. Вид окна приложения TRIGFUNC с недостаточной шириной столбцов

Исправление. Поскольку проблема возникла после добавления в метод Window_Loaded вызова, связанного с отображением окна-заставки, возникает мысль *отсоединить* на это время обработчик Window_LayoutUpdated от события LayoutUpdated, чтобы обусловленное

этим вызовом событие LayoutUpdated не привело к (неправильному) пересчету размеров столбцов. Итак, добавим в начало метода Window_Loaded оператор

```
LayoutUpdated -= Window_LayoutUpdated;
```

а в конец этого же метода – оператор

```
LayoutUpdated += Window_LayoutUpdated;
```

Результат. Теперь главное окно отображается со столбцами правильной ширины, однако неожиданно возникает новая проблема.

Недочет 3. Главное окно неверно центрируется по горизонтали. Не доискиваясь до причин этого недочета, попытаемся его исправить простейшим способом. У нас имеется обработчик Window_LayoutUpdated, в котором правильно устанавливаются размеры столбцов, а вместе с ними и новые размеры окна. Используя эти новые размеры окна, мы можем *самостоятельно* обеспечить его центрирование.

Исправление. Добавьте в конец метода Window_LayoutUpdated следующий оператор:

```
Left = (SystemParameters.WorkArea.Width - ActualWidth) / 2;
```

Указанное добавление еще не исправляет недочет, однако причины этого понятны: для того чтобы обновилось значение свойства ActualWidth, используемое в добавленном операторе, необходимо, чтобы *еще раз* сработало событие LayoutUpdated, учитывающие те изменения размеров, которые мы сделали в методе Window_LayoutUpdated.

Завершающая часть исправления. Добавьте в класс MainWindow вспомогательный метод

```
public static void DoEvents()
{
    Application.Current.Dispatcher.Invoke(System.Windows
      .Threading.DispatcherPriority.Background,
      new Action(delegate { }));
}
```

и вызовите его *после* оператора, отсоединяющего обработчик от события, но *перед* оператором изменения свойства Left, который был ранее добавлен в метод Window_LayoutUpdated:

```
LayoutUpdated -= Window_LayoutUpdated;
DoEvents();
Left = (SystemParameters.WorkArea.Width - ActualWidth) / 2;
```

Результат. Теперь центрирование выполняется правильно.

2.3. Отображение индикатора прогресса при загрузке программы

Рис. 11. Макет окна Window1 приложения TRIGFUNC (второй вариант)

```
<Window x:Class="TRIGFUNC.Window1"
    … >
  <Grid x:Name="grid1" >
    <Ellipse … >
        …
    </Ellipse>
    <TextBlock … >
    </TextBlock>
    <ProgressBar x:Name="progressBar1"
        HorizontalAlignment="Center" Margin="0,50"
        Height="10" VerticalAlignment="Bottom" Width="300"
        Background="#00000000" Foreground="Blue" />
  </Grid>
</Window>
```

Измените цикл в методе Window_Loaded для класса MainWindow следующим образом:

```
for (int i = 0; i < n; i++)
{
    win1.progressBar1.Value = 100 * i / (n - 1);
    tf.Add(new TFData(Math.PI * i * step));
}
```

Результат. Внесенные изменения предназначены для отображения в окне-заставке *индикатора прогресса* (компонент ProgressBar), показывающего, какая часть вычислений выполнена к текущему моменту.

Ошибка 1. Даже для максимального количества точек, равного 1000001, состояние индикатора прогресса на экране не изменяется. Данная

ошибка связана с тем, что изменение индикатора прогресса связано с его перерисовкой, которая не выполняется, пока не произойдет выход из метода Window_Loaded.

Исправление. Добавьте в цикл вызов метода DoEvents:

```
for (int i = 0; i < n; i++)
{
    win1.progressBar1.Value = 100 * i / (n - 1);
    DoEvents();
    tf.Add(new TFData(Math.PI * i * step));
}
```

Ошибка 2. Теперь индикатор прогресса обновляется, но время работы метода Window_Loaded существенно увеличилось. Это связано с тем, что вызов DoEvents требует определенного времени на выполнение, а в данном варианте программы он выполняется на *каждой* итерации цикла.

Исправление. Добавьте в цикл условный оператор:

```
for (int i = 0; i < n; i++)
{
    if (win1.progressBar1.Value != 100 * i / (n - 1))
    {
        win1.progressBar1.Value = 100 * i / (n - 1);
        DoEvents();
    }
    tf.Add(new TFData(Math.PI * i * step));
}
```

Результат. Теперь вызов метода DoEvents выполняется только на тех итерациях, на которых *изменяется* свойство Value компонента ProgressBar (т. е. когда действительно требуется выполнить его перерисовку).

3. Создание компонентов во время выполнения программы: HTOWERS

Рис. 12. Окно приложения HTOWERS

Программа, которая будет разработана в данном примере, представляет собой компьютерную реализацию известной логической задачи «Ханойские башни». Эта задача состоит в следующем: имеются три колышка, на один из которых нанизано (в порядке уменьшения размера) несколько дисков, образующих «башню»; требуется переместить всю башню на один из пустых колышков, пользуясь другим пустым колышком как вспомогательным. Переносить можно по одному диску, причем *больший диск нельзя помещать на меньший*. Заметим, что для решения задачи с N дисками минимальное число переносов равно $2^N–1$ (см., например, [6, гл. 1]).

В нашей программе вместо колышков будут использоваться три прямоугольные области (компоненты DockPanel), а вместо дисков между этими областями будут перемещаться прямоугольные блоки (компоненты Rectangle).

3.1. Настройка начальной позиции

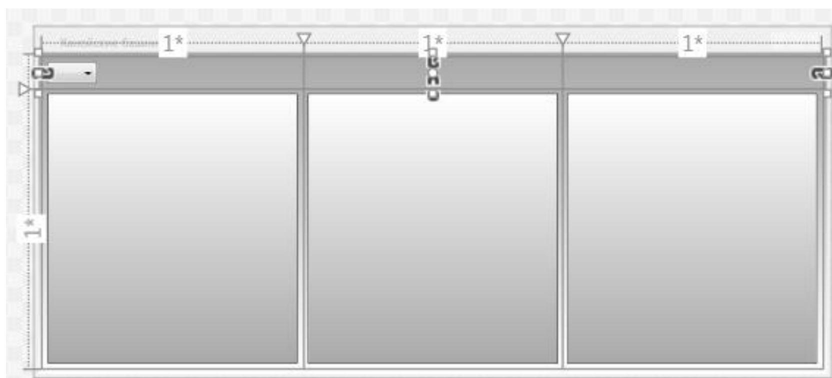


Рис. 13. Макет окна приложения HTOWERS

```
<Window x:Class="HTOWERS.MainWindow"
    …
    Title="Ханойские башни" Height="350" Width="800"
    WindowStartupLocation="CenterScreen" Loaded="Window_Loaded" >
  <Window.Resources>
    <LinearGradientBrush x:Key="GrayBrush" EndPoint="0.5,1"
        StartPoint="0.5,0" >
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="DarkGray" Offset="1" />
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="GrayBrush1" EndPoint="0.5,1"
        StartPoint="0.5,0" >
      <GradientStop Color="DarkGray" Offset="0" />
      <GradientStop Color="White" Offset="1" />
    </LinearGradientBrush>
  </Window.Resources>
  <Window.Background>
    <StaticResource ResourceKey="GrayBrush1"/>
  </Window.Background>
  <Grid Margin="5" >
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel Grid.ColumnSpan="3" Orientation="Horizontal" >
      <ComboBox x:Name="comboBox1" Width="50" Margin="5"
          SelectionChanged="comboBox1_SelectionChanged" />
    </StackPanel>
    <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1"
        Margin="5">
      <DockPanel x:Name="panel1" LastChildFill="False"
          Background="{StaticResource ResourceKey=GrayBrush}" />
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1"
        Grid.Column="1" Margin="5">
```

```
      <DockPanel x:Name="panel2" LastChildFill="False"
          Background="{StaticResource ResourceKey=GrayBrush}" />
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1"
        Grid.Column="2" Margin="5">
      <DockPanel x:Name="panel3" LastChildFill="False"
          Background="{StaticResource ResourceKey=GrayBrush}" />
    </Border>
  </Grid>
</Window>
```

В класс MainWindow добавьте два поля:

```
Random r = new Random();
int total;
```

В конструктор класса MainWindow добавьте два оператора:

```
comboBox1.ItemsSource = Enumerable.Range(2, 9)
  .Select(e => e.ToString());
comboBox1.Focus();
```

И определите обработчики события Loaded для окна и события SelectionChanged для компонента combobox1 (эти обработчики уже указаны в xaml-файле):

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    comboBox1.SelectedIndex = 8;
}
private void comboBox1_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
    total = int.Parse(comboBox1.SelectedItem.ToString());
    panel1.Children.Clear();
    panel2.Children.Clear();
    panel3.Children.Clear();
    var w = (panel1.ActualWidth - 20) / (2 * total);
    for (int i = 0; i < total; i++)
    {
        var r = new Rectangle();
        r.Width = panel1.ActualWidth - (20 + i * 2 * w);
        r.Height = (panel1.ActualHeight - 10) / total;
        r.Stroke = Brushes.Black;
        r.StrokeThickness = 1;
        DockPanel.SetDock(r, Dock.Bottom);
        LinearGradientBrush b = new LinearGradientBrush();
```

```
        b.StartPoint = new Point(0.5, 0);
        b.EndPoint = new Point(0.5, 0.5);
        Color c1 = Color.FromRgb((byte)rnd.Next(256),
          (byte)rnd.Next(256), (byte)rnd.Next(256));
        b.GradientStops.Add(new GradientStop(Colors.White, 0));
        b.GradientStops.Add(new GradientStop(c1, 1));
        r.Fill = b;
        r.RadiusX = r.Height / 8;
        r.RadiusY = r.Height / 8;
        panel1.Children.Add(r);
    }
}
```

Результат. С помощью выпадающего списка comboBox1 можно выбирать количество блоков (от двух до десяти). При запуске программы устанавливается количество блоков, равное десяти, и эти блоки изображаются на первой панели (рис. 76). Обратите внимание на то, что размеры окна можно изменять; при этом изменяются размеры панелей, однако сохраняются размеры «старых» блоков (созданных до изменения размеров окна).

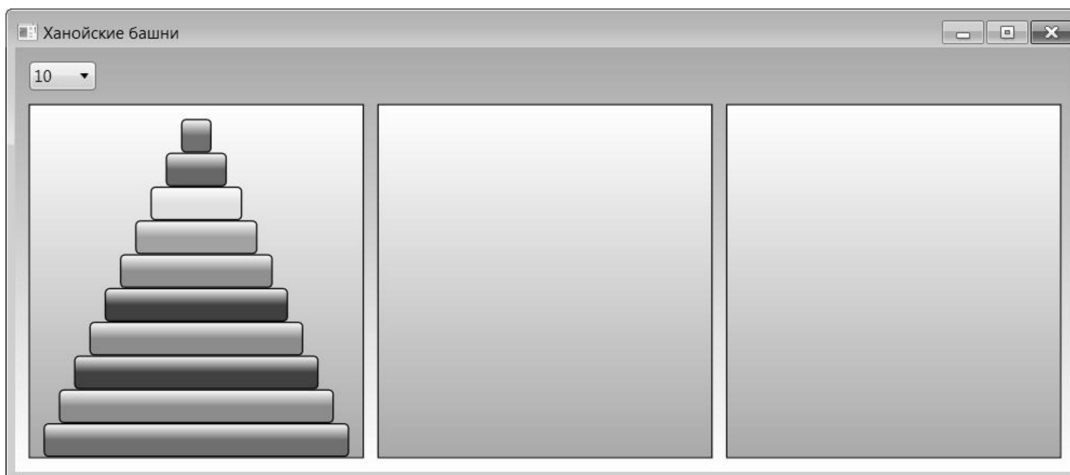


Рис. 14. Окно приложения HTOWERS после запуска

3.2. Перетаскивание блоков на новое место

Для каждого элемента DockPanel в xaml-файле добавьте новый атрибут:

```
AllowDrop="True"
```

В конец цикла for метода combobox1_SelectionChanged добавьте оператор:

```
r.MouseDown += R_MouseDown;
```

Заметим, что после ввода начальной части этого оператора r.MouseDown += появится приглашение завершить ввод, нажав клавишу

Tab. Если это сделать, то будет не только завершен ввод данного оператора, но и создана заготовка для обработчика R_MouseDown следующего вида:

```
private void R_MouseDown(object sender, MouseButtonEventArgs e)
{
    throw new NotImplementedException();
}
```

Измените этот обработчик:

```
private void R_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (e.ChangedButton == MouseButton.Left)
        DragDrop.DoDragDrop(sender as Rectangle, sender,
          DragDropEffects.Move);
}
```

В классе MainWindow определите два вспомогательных метода:

```
DockPanel GetPanel(object trg)
{
    var panel = trg as DockPanel;
    if (panel != null)
        return panel;
    var r = trg as Rectangle;
    if (r != null)
        return r.Parent as DockPanel;
    return null;
}
void MoveRect(Rectangle r, DockPanel panel)
{
    (r.Parent as DockPanel).Children.Remove(r);
    panel.Children.Add(r);
}
```

Используя xaml-файл, определите для компонента panel1 обработчики событий DragEnter и Drop:

```
<DockPanel x:Name="panel1" … DragEnter="panel1_DragEnter"
    Drop="panel1_Drop" />
```

```
private void panel1_DragEnter(object sender, DragEventArgs e)
{
    var panel = GetPanel(e.Source);
    if (panel == null)
        return;
    var r = e.Data.GetData(typeof(Rectangle)) as Rectangle;
```

```
    var oldPanel = r.Parent as DockPanel;
    e.Effects = oldPanel.Children.IndexOf(r) ==
      oldPanel.Children.Count - 1 ?
      DragDropEffects.Move : DragDropEffects.None;
    e.Handled = true;
}
private void panel1_Drop(object sender, DragEventArgs e)
{
    var panel = GetPanel(e.Source);
    if (panel == null)
        return;
    var r = e.Data.GetData(typeof(Rectangle)) as Rectangle;
    if (panel == r.Parent)
        return;
    rect_Move(r, panel);
}
```

После создания обработчиков *переместите* в xaml-файле связанные с ними атрибуты DragEnter="panel1_DragEnter" и Drop="panel1_Drop" в элемент Grid и дополнительно укажите в элементе Grid атрибут, связывающий событие DragOver с уже имеющимся обработчиком panel1_DragEnter:

```
<Grid Margin="5" DragEnter="panel1_DragEnter"
    DragOver="panel1_DragEnter" Drop="panel1_Drop" >
  …
  <Border … >
    <DockPanel x:Name="panel1" … DragEnter="panel1_DragEnter"
        Drop="panel1_Drop" />
```

Результат. Верхний блок (компонент Rectangle) любой башни можно переместить на другую панель DockPanel, причем перемещенный блок всегда будет располагаться на вершине башни. Блоки, расположенные под верхним блоком, перемещать нельзя.

Осталось учесть *дополнительное условие задачи*: блок может перемещаться либо на пустое место, либо на башню с верхним блоком *большего* размера. Для этого откорректируйте метод panel1_DragEnter:

```
private void panel1_DragEnter(object sender, DragEventArgs e)
{
    var panel = GetPanel(e.Source);
    if (panel == null)
        return;
    var r = e.Data.GetData(typeof(Rectangle)) as Rectangle;
    var oldPanel = r.Parent as DockPanel;
```

```
    var c = panel.Children.Count;
    var k = c > 0 ? (panel.Children[c - 1] as Rectangle).Width :
      double.MaxValue;
    e.Effects = oldPanel.Children.IndexOf(r) ==
      oldPanel.Children.Count - 1 && r.Width <= k ?
      DragDropEffects.Move : DragDropEffects.None;
    e.Handled = true;
}
```

Результат. При перемещении блока учитывается дополнительное условие.

3.3. Восстановление начальной позиции, подсчет числа перемещений блоков и контроль за решением задачи

```
<Window x:Class="HTOWERS.MainWindow" … >
  …
    <StackPanel Grid.ColumnSpan="3" Orientation="Horizontal" >
      <ComboBox x:Name="comboBox1" Width="50" Margin="5"
          SelectionChanged="comboBox1_SelectionChanged" />
      <Button x:Name="button1" Content="Сброс" Width="75"
          Margin="5" Click="button1_Click" />
      <TextBlock x:Name="label1" Text="" Margin="5"
          VerticalAlignment="Center" />
      <TextBlock x:Name="label2" Text="Задача решена!" Margin="5"
          Foreground="LightGreen" VerticalAlignment="Center" />
    </StackPanel>
  …
</Window>
```

В описание класса MainWindow добавьте новые поля

```
int count;
int minCount;
```

и вспомогательный метод:

```
void Info()
{
    label1.Text = string.Format("Число перемещений: {0} ({1})",
      count, minCount);
}
```

Перед циклом for в методе comboBox1_SelectionChanged добавьте новые операторы:

```
count = 0;
minCount = (int)Math.Round(Math.Pow(2, total)) - 1;
Info();
```

```
label2.Visibility = Visibility.Hidden;
```

В метод MoveRect добавьте следующие операторы:

```
count++;
Info();
if (panel2.Children.Count == total ||
  panel3.Children.Count == total)
    label2.Visibility = Visibility.Visible;
```

И определите обработчик события Click для кнопки button1, указанный в xaml-файле:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    comboBox1_SelectionChanged(null, null);
}
```

Результат. 1. Для восстановления начальной позиции с тем же количеством блоков следует нажать кнопку «Сброс». Если при восстановлении начальной позиции требуется изменить число блоков, то по-прежнему достаточно указать новое значение в компоненте comboBox1 (нажимать кнопку «Сброс» в этой ситуации не требуется).

2. Информация о числе перемещений блоков выводится в тексте метки label1. Там же, в скобках, указывается количество перемещений, минимально необходимое для решения задачи с данным числом блоков n (это количество равно $2^n - 1$).

3. Задача считается решенной (и об этом выводится сообщение на экран), если размер башни в конечной позиции (на панели panel2 или panel3) равен общему числу блоков.

Недочет. После решения задачи по-прежнему разрешено перемещение блоков.

Исправление. Добавьте в начало метода panel1_DragEnter следующий фрагмент:

```
if (label2.IsVisible)
{
    e.Effects = DragDropEffects.None;
    e.Handled = true;
    return;
}
```

Результат. Теперь после решения задачи блоки нельзя перемещать.

3.4. Демонстрационное решение задачи

```
<Window x:Class="HTOWERS.MainWindow" … >
  …
    <StackPanel Grid.ColumnSpan="3" Orientation="Horizontal" >
```

```
      <ComboBox x:Name="comboBox1" Width="50" Margin="5"
          SelectionChanged="comboBox1_SelectionChanged" />
      <Button x:Name="button1" Content="Сброс" Width="75"
          Margin="5" Click="button1_Click" />
      <Button x:Name="button2" Content="Демо" Width="75"
          Margin="5" Click="button2_Click" />
      …
    </StackPanel>
  …
</Window>
```

В начало метода R_MouseDown добавьте следующий фрагмент:

```
if (!button1.IsEnabled)
    return;
```

В описание класса MainWindow добавьте два вспомогательных метода:

```
public static void DoEvents()
{
    Application.Current.Dispatcher.Invoke(System.Windows
      .Threading.DispatcherPriority.Background,
      new Action(delegate { }));
}
private void Step(int k, DockPanel src, DockPanel dst,
  DockPanel tmp)
{
    if (k == 0)
        return;
    Step(k - 1, src, tmp, dst);
    if (button1.IsEnabled)
        return;
    MoveRect(src.Children[src.Children.Count - 1] as Rectangle,
      dst);
    DoEvents();
    System.Threading.Thread.Sleep(1500 / (total - 1));
    Step(k - 1, tmp, dst, src);
}
```

И определите обработчик события Click для кнопки button2, уже указанный в xaml-файле:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    comboBox1.IsEnabled = button1.IsEnabled = !button1.IsEnabled;
    if (!button1.IsEnabled)
```

```
    {
        if (panel1.Children.Count != total)
            comboBox1_SelectionChanged(null, null);
        Step(total, panel1, panel3, panel2);
        comboBox1.IsEnabled = button1.IsEnabled = true;
    }
}
```

Результат. При нажатии на кнопку «Демо» программа переходит в *демо-режим*, показывающий правильное решение задачи с указанным числом блоков (блоки перемещаются автоматически). В демо-режиме блокируется компонент comboBox1 и кнопка «Сброс»; кроме того, в нем запрещено перетаскивание блоков. Выход из демо-режима происходит после завершения решения задачи, а также при повторном нажатии на кнопку «Демо» (в последнем случае после выхода из демо-режима можно продолжать решать задачу самостоятельно).

Ошибка. При попытке завершить программу, находящуюся в демо-режиме, возникает исключение, связанное с тем, что ранее вызванные методы Step продолжают выполняться уже после того, как компоненты окна были разрушены в результате завершающих действий программы.

Исправление. Определите обработчик события Closed для окна:

```
<Window x:Class="HTOWERS.MainWindow"
    … Closed="Window_Closed" >
```

```
private void Window_Closed(object sender, EventArgs e)
{
    button1.IsEnabled = true;
}
```

Результат. Теперь при закрытии окна кнопка button1 делается доступной, что позволяет практически немедленно завершить все рекурсивные вызовы метода Step благодаря присутствующему в методе Step условному оператору.