Глава 7. Курсоры и значки: проект CURSORS

7.1. Использование стандартных курсоров

После создания проекта CURSORS разместите в форме `Form1` кнопку `button1` и настройте свойства формы и добавленной кнопки (листинг 7.1, рис. 7.1).

В начало описания класса `Form1` добавьте описания двух полей:

```
private List<string> str = new List<string>(30);
private List<Cursor> cur = new List<Cursor>(30);
```

Добавьте новые операторы в конструктор класса `Form1` (листинг 7.2) и определите обработчик события `MouseDown` для кнопки `button1` (листинг 7.3).

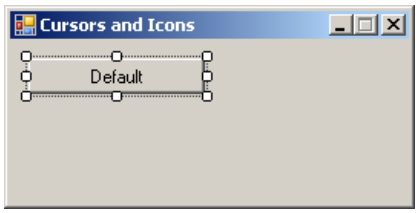

Рис. 7.1. Вид формы Form1 для проекта CURSORS на начальном этапе разработки

Листинг 7.1. Настройка свойств

```
Form1: Text = Cursors and Icons,
  MaximizeBox = False,
  FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen
button1: Text = Default
```

Листинг 7.2. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  foreach (System.Reflection.PropertyInfo pi in
    typeof(Cursors).GetProperties())
  {
    str.Add(pi.Name);
    cur.Add((Cursor)pi.GetValue(null, null));
  }
  button1.Tag = str.IndexOf("Default");
}
```

Листинг 7.3. Обработчик `button1.MouseDown`

```
private void button1_MouseDown(object sender, MouseEventArgs e)
{
  int k = (int)button1.Tag,
    c = str.Count;
  switch (e.Button)
  {
    case MouseButtons.Left:
      k = (k + 1) % c; break;
    case MouseButtons.Right:
      k = (k - 1 + c) % c; break;
  }
  button1.Text = str[k];
  button1.Cursor = cur[k];
  button1.Tag = k;
}
```

Результат: нажатие мышью на кнопку **Default** приводит к смене курсора для данной кнопки; при этом заголовок кнопки заменяется на имя текущего курсора. 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.

7.2. Установка курсора для формы и индикация режима ожидания с помощью курсора

Разместите в форме `Form1` кнопку три новые кнопки `button2`, `button3`, `button4` и присвойте свойствам `Text` этих кнопок значения **`Form Cursor`**, **`Wait Cursor`** и **`Default Form Cursor`** соответственно (рис. 7.2). Определите обработчики события `Click` для добавленных кнопок (листинг 7.4).

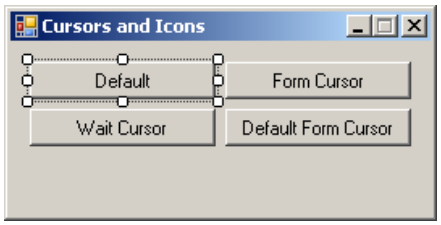

Рис. 7.2. Вид формы Form1 для проекта CURSORS на промежуточном этапе разработки

Листинг 7.4. Обработчики `button2.Click`, `button3.Click` и `button4.Click`

```
private void button2_Click(object sender, EventArgs e)
{
  Cursor = button1.Cursor;
}
private void button3_Click(object sender, EventArgs e)
{
  UseWaitCursor = true;
}
private void button4_Click(object sender, EventArgs e)
{
  UseWaitCursor = false;
  Cursor = Cursors.Default;
}
```

Результат: нажатие на кнопку **Form Cursor** распространяет действие нового курсора на всю форму, включая находящиеся на ней кнопки. Нажатие на кнопку **Wait Cursor** устанавливает для всей формы и ее компонентов *курсор ожидания* (WaitCursor) в виде песочных часов. Нажатие на кнопку **Default Form Cursor** отменяет курсор ожидания и восстанавливает стандартный курсор для формы (но не для первой кнопки, курсор которой по-прежнему соответствует названию, приведенному в ее заголовке).

7.3. Подключение к проекту новых курсоров и их сохранение в виде внедренных ресурсов

Кроме стандартных курсоров в программе можно использовать дополнительные курсоры. Файлы с курсорами имеют расширение cur; их можно найти, например, в подкаталоге Cursors каталога Windows. Следует отметить, что *анимированные* курсоры (файлы с расширением ani) использовать в Visual C# нельзя, а *цветные* курсоры при использовании в программе отображаются в монохромном режиме. Значок любого cur-файла соответствует изображению курсора, содержащегося в этом файле; это позволяет быстро ознакомиться с содержимым cur-файлов, используя приложения **Проводник** или **Мой компьютер**.

Выберите два каких-либо cur-файла, содержащих монохромные курсоры, и скопируйте их в каталог проекта CURSORS под именами C1.cur и C2.cur.

Загрузите в среду Visual C# проект CURSORS, если это еще не сделано.

Выполните команду меню **Project | Add Existing Item…**. В появившемся диалоговом окне введите имя файла **C1.cur** (это можно сделать 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, в чем можно убедиться, посмотрев на окно **Solution Explorer**.

Выделите файл C1.cur в окне **Solution Explorer**; при этом в окне **Properties** будут отображены его свойства. Установите свойство `Build Action` равным **`Embedded Resource`**; прочие свойства изменять не требуется.

Аналогичными действиями добавьте к проекту файл C2.cur и настройте его свойство `Build Action`.

В конструктор класса `Form1` добавьте новые операторы (листинг 7.5).

Листинг 7.5. Добавление к конструктору класса `Form1`

```
for (int i = 1; i <= 2; i++)
{
  str.Add("C" + i);
  cur.Add(new Cursor(GetType(), "C" + i + ".cur"));
}
```

Результат: при создании формы новые курсоры загружаются в программу и добавляются к списку доступных курсоров под именами `C1` и `C2` . В дальнейшем они обрабатываются аналогично стандартным (см. разд. 7.1 и 7.2). Следует подчеркнуть, что изображения этих курсоров размещаются непосредственно в исполняемом файле приложения CURSORS.exe.

7.4. Работа со значками

Небольшие изображения, называемые *значками*, *иконками*, или *пиктограммами*, хранятся в файлах с расширением .ico. Большое количество ico-файлов содержится в библиотеке изображений, входящей в состав системы Visual Studio 2005 (библиотека поставляется в виде архива VS2005ImageLibrary.zip, который размещается в подкаталоге Common7\VS2005ImageLibrary каталога Visual Studio). Аналогичная библиотека с именем VS2008ImageLibrary входит в состав системы Visual Studio 2008. В экспресс-варианты Visual C# библиотека изображений не входит, однако найти несколько

ico-файлов на компьютере, как правило, не составляет труда. Значок ico-файла, подобно значку cur-файла, соответствует содержащемуся в нем изображению.

Добавление к проекту существующих ico-файлов выполняется аналогично добавлению cur-файлов. При описании последующих действий предполагается, что к проекту добавлены файлы Computer.ico и Folder.ico (в библиотеке VS2005ImageLibrary они находятся в подкаталоге icons\Misc, а в библиотеке VS2008ImageLibrary — в подкаталоге Objects\ico_format\Office & Dev). Как и в случае cur-файлов, после добавления ico-файла в проект следует установить его свойство `Biuld Action` равным **`Embedded Resource`**.

Разместите в форме кнопку `button5` и присвойте ее свойству `Text` значение **`Icon 0`**. В начало описания класса `Form1` добавьте описание массива `ico`:

```
private Icon[] ico = new Icon[3];
```

Добавьте в конструктор класса `Form1` новые операторы (листинг 7.6) и определите обработчик события `Click` для кнопки `button5` (листинг 7.7).

Листинг 7.6. Добавление к конструктору класса `Form1`

```
button5.Tag = 0;
ico[1] = new Icon(GetType(), "Computer.ico");
ico[2] = new Icon(GetType(), "Folder.ico");
ico[0] = Icon;
```

Листинг 7.7. Обработчик `button5.Click`

```
private void button5_Click(object sender, EventArgs e)
{
  int k = ((int)button5.Tag + 1) % 3;
  button5.Text = "Icon " + k;
  button5.Tag = k;
  Icon = ico[k];
}
```

Результат: кнопка `button5` изменяет значок формы как в ее заголовке, так и на ее кнопке, находящейся на панели задач (в нижней части экрана). Значок с номером 0 соответствует исходному значку формы, присвоенному ей по умолчанию. Для хранения трех значков в программе используется массив `ico` фиксированного размера.

7.5. Размещение значка в области уведомлений

Разместите в форме еще одну кнопку (`button6`), а также компонент `NotifyIcon` (этот компонент получит имя `notifyIcon1`), и настройте их свойства (листинг 7.8, рис. 7.3).

Заметим, что компонент `NotifyIcon` не является визуальным компонентом, поэтому при его добавлении в форму он размещается в специальной *области невизуальных компонентов*, расположенной под изображением формы.

В конструкторе класса `Form1` дополните последний оператор следующим образом:

```
notifyIcon1.Icon = ico[0] = Icon;
```

В методе `button5_Click` также дополните последний оператор:

```
notifyIcon1.Icon = Icon = ico[k];
```

Определите обработчик события `Click` для кнопки `button6` (листинг 7.9).

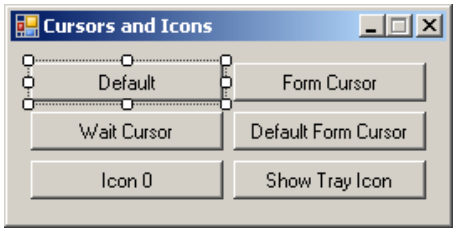

Рис. 7.3. Окончательный вид формы Form1 для проекта CURSORS

Листинг 7.8. Настройка свойств

```
button6: Text = Show Tray Icon
notifyIcon1: Text = Icon in Traybar,
  Visible = False
```

Листинг 7.9. Обработчик `button6.Click`

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

}

Результат: нажатие на кнопку `button6` скрывает кнопку формы на панели задач и отображает ее значок в *области уведомлений* (notification area, traybar) в правой части панели задач. При наведении курсора на этот значок возникает всплывающая подсказка **Icon in Traybar**. Повторное нажатие кнопки `button6` восстанавливает исходное представление формы на панели задач.

Глава 30. Приложение с заставкой: проект TRIGFUNC

30.1. Формирование таблицы значений тригонометрических функций

После создания проекта TRIGFUNC разместите в форме `Form1` компонент-таблицу dataGridView1 и настройте свойства формы и добавленного компонента (листинг 5.1).

Кроме настройки свойств таблицы `dataGridView1` "в целом", необходимо настроить свойства ее столбцов. Для этого выберите в окне **Properties** компонента `dataGridView1` свойство `Columns` и нажмите расположенную рядом с ним кнопку с многоточием. На экране появится диалоговое окно **Edit Columns**, позволяющее создавать столбцы таблицы, задавать их порядок и настраивать их свойства. Нажмите в этом окне кнопку **Add**, в появившемся окне **Add Column** введите в поле **Header text** заголовок первого столбца: **`x`** (прочие настройки менять не требуется) и нажмите кнопку **Add**. В левой части окна **Edit Columns** появится изображение созданного столбца, а в правой — таблица, позволяющая настраивать его свойства (структура данной таблицы подобна структуре окна **Property**). Аналогичными действиями добавьте в таблицу `dataGridView1` еще четыре столбца, указав для них следующие заголовки: **`Sin(x*pi)`**, **`Cos(x*pi)`**, **`Tg(x*pi)`**, **`Ctg(x*pi)`**. После этого закройте окно **Edit Columns**, нажав кнопку **OK**. В результате в компоненте `dataGridView1` будут изображены заголовки созданных столбцов (см. рис. 30.1).

Определите обработчик события `Load` формы `Form1` (листинг 30.2).

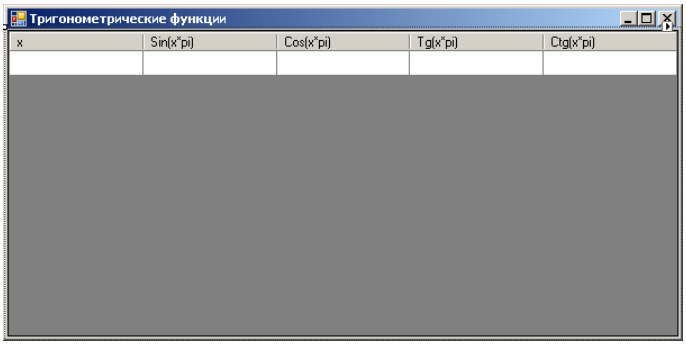

Рис. 30.1. Вид формы Form1 для проекта TRIGFUNC на начальном этапе разработки

Листинг 30.1. Настройка свойств

```
Form1: Text = Тригонометрические функции,
  StartPosition = CenterScreen
dataGridView1: Dock = Fill,
  AllowUserToResizeColumns = False,
  AllowUserToResizeRows = False,
  AutoSizeColumnsMode = Fill,
  RowHeadersVisible = False, ReadOnly = True
```

Листинг 30.2. Обработчик `Form1.Load`

```
private void Form1_Load(object sender, EventArgs e)
{
  int n = 7,
    nMax = 100001;
  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", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
      Close();
```

```
      return;
    }
  double step = 1.0 / (n - 1);
  dataGridView1.Columns[0].Width /= 2;
          // - уменьшаем ширину первого столбца
  dataGridView1.RowCount = n;
          // - определяем количество строк
  for (int i = 0; i < n; i++)
  {
    double x = i * step,
      sx = Math.Sin(Math.PI * x),
      cx = Math.Cos(Math.PI * x);
    dataGridView1[0, i].Value = x.ToString("f5");
    dataGridView1[1, i].Value = sx.ToString();
    dataGridView1[2, i].Value = cx.ToString();
    dataGridView1[3, i].Value = (sx / cx).ToString();
    dataGridView1[4, i].Value = (cx / sx).ToString();
  }
}
```

Результат: после запуска программа заполняет таблицу `dataGridView1` значениями тригонометрических функций с аргументами от 0 до π радианов на сетке из `n` равноотстоящих точек. Число точек `n` можно указать в качестве параметра командной строки; допускаются значения от 2 до 100001. При запуске программы из среды Visual Studio параметры можно указать в поле **Command line arguments** группы настроек **Debug** в окне свойств проекта (напомним, что это окно вызывается командой **Project |** *<Имя проекта>* **Properties**). Если параметр указан неверно, то выдается сообщение об ошибке и программа немедленно завершает работу. Если параметр не указан, то количество точке полагается равным 7.

30.2. Отображение окна-заставки при загрузке программы

Добавьте к проекту новую форму (она автоматически получит имя `Form2`) и разместите в форме `Form2` метку `label1`. Настройте свойства формы `Form2` и метки `label1` (листинг 30.3). Для настройки указанных свойств (`Name`, `Size` и `Bold`) свойства `Font` метки `label1` удобно вывести на экран диалоговое окно **Шрифт**, нажав на кнопку с многоточием рядом со свойством `Font` в окне **Properties**. С помощью окна **Шрифт** можно настроить все свойства шрифта и, кроме того, увидеть образец шрифта с выбранными настройками.

После настройки свойства `Font` компонента `label1` измените размер формы `Form2` так, чтобы текст метки размещался в двух строках (см. рис. 30.2).

> ***Примечание***
> Напомним, что для выбора формы в ситуации, когда выбран один из ее компонентов, достаточно нажать клавишу <Esc>. Еще один быстрый способ выбора формы — щелчок мышью на ее заголовке — в данном случае неприменим, так как сделанные выше настройки свойств формы привели к тому, что заголовок в ней отсутствует.

В начало описания класса `Form1` в файле Form1.cs добавьте новое поле

```
private Form2 form2 = new Form2();
```

Дополните конструктор класса `Form1` (листинг 30.4).

В конец метода `Form1_Load` (листинг 30.2) добавьте оператор

```
form2.Hide();
```

Рис. 30.2. Вид формы Form2 для проекта TRIGFUNC на начальном этапе разработки

Листинг 30.3. Настройка свойств формы `Form2` и ее компонентов

```
Form2: Text = пустая строка, ControlBox = False,
  FormBorderStyle = FixedSingle, Opacity = 80%,
  ShowInTaskbar = False,
  StartPosition = CenterScreen,
  UseWaitCursor = True
label1: Text = Тригонометрические функции,
  AutoSize = False, Dock = Fill,
  TextAlign = MiddleCenter,
  Font.Name = Times New Roman, Font.Size = 32,
```

```
  Font.Bold = True
```

Листинг 30.4. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
  AddOwnedForm(form2);
  form2.Show();
  Application.DoEvents();
}
```

Результат: в ходе загрузки главной формы Form1 и заполнения таблицы dataGridView1 данными на экране отображается окно-заставка Form2, свидетельствующая о том, что программа запущена и в настоящий момент выполняет инициализирующие действия. Заставка не имеет заголовка и является полупрозрачной; ее нельзя перемещать по экрану. Курсор мыши на заставке принимает вид песочных часов. При отображении главной формы заставка исчезает.

Недочет: если объем вычислений, необходимый для заполнения таблицы, невелик (например, при использовании числа точек n = 7, заданного по умолчанию), то рассмотреть заставку не удается из-за краткого времени ее отображения.

Исправление: в методе Form1_Load перед добавленным ранее оператором

```
form2.Hide();
```

вставьте еще один оператор:

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

Результат: после завершения инициализирующих действий программа приостанавливает работу на 1 секунду, в течение которой окно-заставка остается на экране.

30.3. Использование окна-заставки в качестве информационного окна

Разместите в форме Form2 кнопку button1 и настройте свойства формы Form2 и добавленного компонента (листинг 30.5; см. также рис. 30.3).

Разместите в форме Form1 компонент-меню menuStrip1. Если добавленная к форме Form1 строка меню будет заслонять верхнюю часть таблицы dataGridView1, то выделите компонент menuStrip1 и выполните для него команду **Send to Back**, нажав вторую справа кнопку на панели **Layout**.

Используя конструктор меню (см. разд. 18.1), создайте в меню menuStrip1 пункт меню первого уровня с текстом **&About**... (см. рис. 30.4) и с помощью окна **Properties** измените имя этого пункта (то есть свойство Name) на **about1**. Определите обработчик события Click для созданного пункта меню (листинг 30.6).

В конец метода Form1_Load добавьте два оператора:

```
form2.UseWaitCursor = false;
form2.Controls["button1"].Visible = true;
```

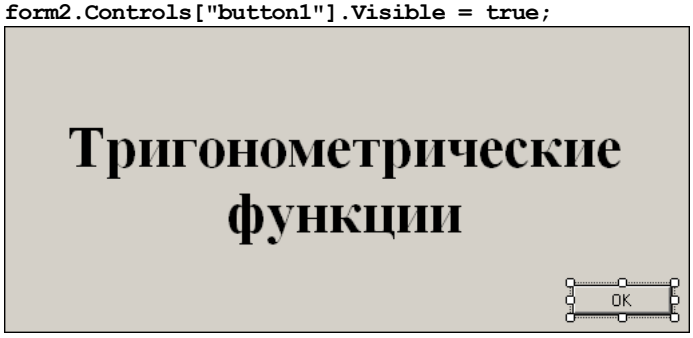

Рис. 30.3. Вид формы Form2 для проекта TRIGFUNC на промежуточном этапе разработки

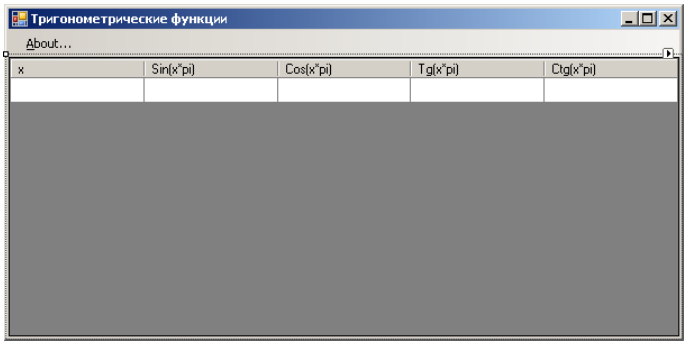

Рис. 30.4. Окончательный вид формы Form1 для проекта TRIGFUNC

Листинг 30.5. Настройка свойств формы Form2 и ее компонентов

```
Form2: AcceptButton = button1
button1: Text = OK, DialogResult = OK,
```

```
  Visible = False
```

Листинг 30.6. Обработчик about1.Click

```
private void about1_Click(object sender, EventArgs e)
{
  form2.ShowDialog();
}
```

Результат: теперь окно-заставку можно выводить на экран (в виде модального окна) после загрузки главной формы, используя команду меню **About…** . Для закрытия окна-заставки в этом случае предназначена кнопка **OK**. При выводе окна-заставки в ходе загрузки главной формы кнопка **OK** на нем не отображается.

30.4. Вывод в окне-заставке информации о ходе загрузки программы

Разместите в форме Form2 компонент типа ProgressBar (он получит имя progressBar1) и настройте его свойства (листинг 30.7; см. также рис. 30.5).

Откорректируйте метод Form1_Load формы Form1 (листинг 30.8).

Рис. 30.5. Окончательный вид формы Form2 для проекта TRIGFUNC

Листинг 30.7. Настройка свойств

```
progressBar1: Dock = Bottom, Modifiers = Internal,
  Style = Continuous
```

Листинг 30.8. Новый вариант метода Form1_Load

```
private void Form1_Load(object sender, EventArgs e)
{
  int n = 7,
    nMax = 100001;
  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", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
      Close();
      return;
    }
  double step = 1.0 / (n - 1);
  dataGridView1.Columns[0].Width /= 2;
  dataGridView1.RowCount = n;
  for (int i = 0; i < n; i++)
  {
    double x = i * step,
      sx = Math.Sin(Math.PI * x),
      cx = Math.Cos(Math.PI * x);
    dataGridView1[0, i].Value = x.ToString("f5");
    dataGridView1[1, i].Value = sx.ToString();
    dataGridView1[2, i].Value = cx.ToString();
    dataGridView1[3, i].Value = (sx / cx).ToString();
    dataGridView1[4, i].Value = (cx / sx).ToString();
    form2.progressBar1.Value = 100 * i / (n - 1);
```

```
  }
  System.Threading.Thread.Sleep(1000);
  form2.Hide();
  form2.UseWaitCursor = false;
  form2.Controls["button1"].Visible = true;
  form2.progressBar1.Visible = false;
}
```

Результат: в ходе загрузки главной формы в окне-заставке выводится графическая информация о доле выполненных вычислений (для этого используется компонент `progressBar1`). При последующих вызовах окна-заставки командой **About…** компонент `progressBar1` на нем не отображается.

30.5. Досрочное завершение программы

Определите обработчик события `KeyDown` формы `Form2` (листинг 30.9).

В методе `Form1_Load` формы `Form1` (листинг 30.8) в начало цикла for вставьте новый фрагмент:

```
Application.DoEvents();
if (!form2.Visible)
{
  Close();
  return;
}
```

Листинг 30.9. Обработчик `Form2.KeyDown`

```
private void Form2_KeyDown(object sender, KeyEventArgs e)
{
  if (e.KeyCode == Keys.Escape)
    Hide();
}
```

Результат: теперь выполнение программы можно прервать в процессе выполнения начальных вычислений; для этого достаточно нажать клавишу <Esc>. При этом окно-заставка исчезает с экрана, и программа немедленно завершает работу.

30.6. Возможность перемещения окна-заставки

В описание класса `Form2` добавьте поле

```
private Point p;
```

Определите обработчики событий `MouseDown` и `MouseMove` для компонента `label1` формы `Form2` (листинг 30.10).

Листинг 30.10. Обработчики `label1.MouseDown` и `label1.MouseMove` (форма `Form2`)

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
}
private void label1_MouseMove(object sender, MouseEventArgs e)
{
  if (Control.MouseButtons == MouseButtons.Left)
    Location += new Size(e.X - p.X, e.Y - p.Y);
}
```

Результат: окно-заставку теперь можно перемещать по экрану с помощью мыши; для этого надо установить курсор мыши в любой позиции окна, нажать левую кнопку мыши и, не отпуская ее, переместить мышь в требуемую позицию экрана.

> ***Примечание***
> Ранее подобный способ перетаскивания был использован в гл. 10. Этот способ является наиболее естественным для тех форм, которые не содержат заголовка.

Недочет: окно-заставку можно перетаскивать и при его первом отображении на экране (в момент инициализации данных), что обычно является нежелательным. **Исправление**: выделите компонент `label1` и в окне **Properties** *очистите* поле, связанное с событием `MouseMove` этого компонента. В описании метода `label1_MouseMove` замените модификатор `private` на **`internal`**. В файле Form1.cs в конце метода `Form1_Load` добавьте оператор:

```
form2.Controls["label1"].MouseMove += form2.label1_MouseMove;
```

Результат: теперь перетаскивание окна-заставки возможно только при его отображении на экране командой **About…** . В момент начальной загрузки программы окно-заставку перетаскивать нельзя, поскольку в этот момент метод `label1_MouseMove` еще не связан с событием `MouseMove` компонента `label1`.

Приведем изображение работающей программы после вызова окна-заставки командой **About…** в случае, когда число точек `n` равно 9 (рис. 30.6).

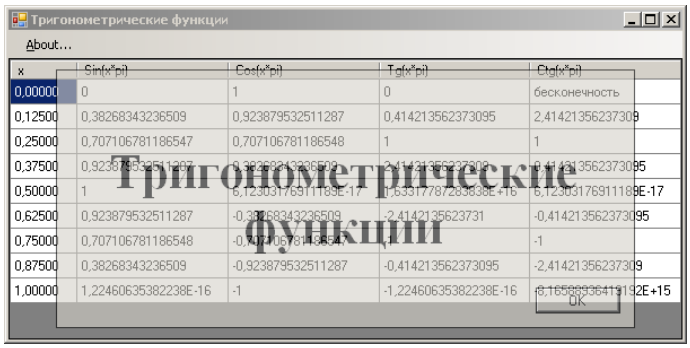

Рис. 30.6. Вид работающего приложения TRIGFUNC

Глава 33. Создание компонентов во время выполнения программы: проект HTOWERS

Проект HTOWERS 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.

33.1. Создание начальной позиции

После создания проекта HTOWERS разместите в форме `Form1` три компонента типа GroupBox (они получат имена `groupBox1`–`groupBox3`) и компонент типа `NumericUpDown` (он получит имя `numericUpDown1`). Настройте свойства формы `Form1` и всех добавленных компонентов (листинг 33.1) и расположите компоненты в соответствии с рис. 33.1.

Определите обработчик события `Load` для формы `Form1` (листинг 33.2).

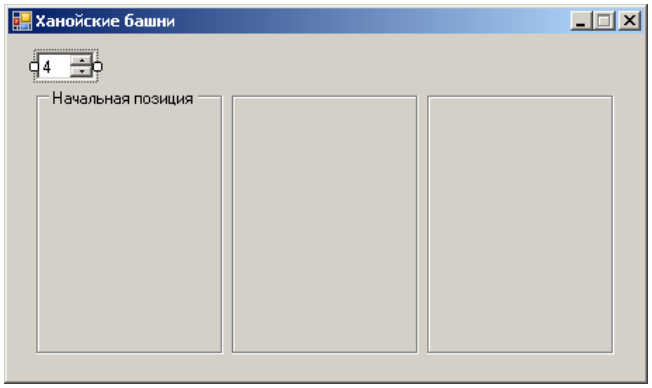

Рис. 33.1. Вид формы Form1 для проекта HTOWERS на начальном этапе разработки

Листинг 33.1. Настройка свойств

```
Form1: Text = Ханойские башни, MaximizeBox = False,
  FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen
groupBox1: Text = Начальная позиция
groupBox2-groupBox2: Text = пустая строка
numericUpDown1: Minimum = 2, Maximum = 10,
  Value = 4
```

Листинг 33.2. Обработчик `Form1.Load`

```
private void Form1_Load(object sender, EventArgs e)
{
  Random r = new Random();
  int n = (int)numericUpDown1.Value,
          // n - количество блоков башни
    w = groupBox1.Width,
    h = groupBox1.Height;
  for (int i = 0; i < n; i++)
  {
    Label lb = new Label();
    // определение свойств созданного компонента
    // Label (i-го блока башни)
    lb.Parent = groupBox1;
```

```
    // - все блоки башни рисуются
    // на компоненте groupBox1
    lb.BorderStyle = BorderStyle.FixedSingle;
    lb.Size = new Size((w - 10) * (n - i) / n, (h - 15) / n);
    lb.Location =
      new Point((w - lb.Width) / 2, h - 2 - (i + 1) * lb.Height);
    lb.BackColor = Color.FromArgb(r.Next(256), r.Next(256), r.Next(256));
    // - цвет определяется случайным образом
  }
}
```

Результат: при запуске программы на компоненте `groupBox1` рисуется башня из четырех разноцветных блоков — компонентов типа `Label`.

33.2. Перерисовка башни при изменении количества блоков

Определите обработчик события ValueChanged для компонента numericUpDown1 (листинг 33.3).

Листинг 33.3. Обработчик `numericUpDown1.ValueChanged`

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
  Form1_Load(this, null);
}
```

Результат: при изменении значения компонента `numericUpDown1` создается новая башня с указанным числом блоков.

Ошибка: новая башня рисуется *под* старой.

Исправление: в класс `Form1` добавьте новый метод `label_Dispose` (листинг 33.4).

В начало метода `numericUpDown1_ValueChanged` добавьте оператор

```
label_Dispose(groupBox1);
```

Листинг 33.4. Метод `label_Dispose` формы `Form1`

```
private void label_Dispose(GroupBox gb)
{
  for (int i = gb.Controls.Count - 1; i >= 0; i--)
    gb.Controls[i].Dispose();
}
```

Результат: теперь все блоки старой башни исчезают с экрана, а также освобождают свои ресурсы (благодаря вызову метода `Dispose`).

Недочет: в процессе формирования новой башни на экране мелькают посторонние фрагменты изображений. Отмеченный недочет связан с тем, что после определения родительского компонента метка немедленно рисуется на нем, а все последующие изменения ее свойств приводят к ее перерисовке.

Исправление: в методе `Form1_Load` (листинг 33.2) переместите оператор

```
lb.Parent = groupBox1;
```

в конец цикла (на позицию после оператора, определяющего фоновый цвет метки).

Результат: теперь все настройки свойств меток выполняются до определения ее родительского компонента, и поэтому не сопровождаются ее перерисовкой на экране (метка рисуется единственный раз после задания всех ее свойств).

33.3. Перетаскивание блоков на новое место

В конструктор класса `Form1` добавьте новый оператор:

```
groupBox1.AllowDrop = groupBox2.AllowDrop = groupBox3.AllowDrop = true;
```

В класс `Form1` добавьте новые методы `label_MouseDown` и `label_Move` (листинг 33.5).

В конец тела цикла метода `Form1_Load` (листинг 33.2) добавьте оператор

```
lb.MouseDown += label_MouseDown;
```

Определите обработчики событий `DragEnter` и `DragDrop` для компонента `groupBox1`, после чего свяжите созданные обработчики с событиями `DragEnter` и `DragDrop` компонентов `groupBox2`– `groupBox3`.

Листинг 33.5. Методы `label_MouseDown` и `label_Move` формы `Form1`

```
private void label_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    DoDragDrop(sender, DragDropEffects.Move);
}
private void label_Move(Label lb, GroupBox gb)
{
  lb.Parent = gb;
  // - смена компонента GroupBox,
  // на котором рисуется метка lb
  lb.Top = gb.Height - 2 - gb.Controls.Count * lb.Height;
```

```
}
```

Листинг 33.6. Обработчики `groupBox1.DragEnter` и `groupBox1.DragDrop`

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void groupBox1_DragDrop(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  GroupBox gb = sender as GroupBox;
  if (gb == lb.Parent)
    return;
  label_Move(lb, gb);
}
```

Результат: любой блок (метку `Label`) можно переместить на другой компонент `GroupBox`, причем перемещенный блок всегда будет располагаться на вершине башни.

Примечание

Действия, связанные с перемещением метки на другой компонент `GroupBox`, оформлены в виде отдельного метода `label_Move` (листинг 33.5), поскольку в дальнейшем они будут выполняться не только при выполнении обработчиков `DragDrop`, но и в особом демо-режиме программы (см. разд. 33.7).

Ошибка: переместить можно не только верхний, но и любой другой блок башни. **Исправление**: измените метод `groupBox1_DragEnter` (листинг 33.7).

Листинг 33.7. Новый вариант метода `groupBox1_DragEnter`

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  if (lb.Parent.Controls[lb.Parent.Controls.Count - 1] != lb)
    e.Effect = DragDropEffects.None;
  else
    e.Effect = DragDropEffects.Move;
}
```

Результат: теперь переместить можно только верхний блок башни; при попытке переместить нижние блоки курсор перетаскивания сразу становится запрещающим.

Примечание

Мы воспользовались тем обстоятельством, что при добавлении дочернего компонента в коллекцию `Controls` он располагается в ее конце; таким образом, верхний элемент башни имеет в коллекции `Controls` индекс, равный числу элементов коллекции `Controls.Count` минус `1`.

Осталось учесть дополнительное условие задачи: блок может перемещаться либо на пустое место, либо на башню с верхним блоком большего размера. Для этого еще раз изменим метод `groupBox1_DragEnter` (листинг 33.8).

Листинг 33.8. Новый вариант метода `groupBox1_DragEnter`

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  int k = int.MaxValue;
  GroupBox gb = sender as GroupBox;
  if (gb.Controls.Count > 0)
    k = gb.Controls[gb.Controls.Count - 1].Width;
  if (lb.Parent.Controls[lb.Parent.Controls.Count - 1] != lb
    || lb.Width > k)
    e.Effect = DragDropEffects.None;
  else
    e.Effect = DragDropEffects.Move;
}
```

Результат: при перемещении блока учитывается дополнительное условие.

Примечание

В новом варианте метода `groupBox1_DragEnter` используется переменная `k`, в которую записывается ширина верхнего блока панели-приемника или максимальное возможное значение типа `int` (равное `int.MaxValue`) если панель-приемник не содержит блоков.

Ошибка: при изменении количества блоков с помощью компонента `numericUpDown1` удаляются только те блоки, которые находятся на панели `groupBox1`.

Исправление: измените метод `numericUpDown1_ValueChanged` (листинг 33.9).

Листинг 33.9. Новый вариант метода `numericUpDown1_ValueChanged`

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
```

```
{
  label_Dispose(groupBox1);
  label_Dispose(groupBox2);
  label_Dispose(groupBox3);
  Form1_Load(this, null);
}
```

33.4. Восстановление начальной позиции и подсчет числа перемещений блоков

Разместите в форме `Form1` кнопку `button1`, положите свойство `Text` кнопки равным строке **`Начальная позиция`** и свяжите ее событие `Click` с существующим обработчиком `numericUpDown1_ValueChanged`. Кроме того, разместите в форме `Form1` метку `label1` (ее свойство `Text` изменять не требуется). Расположите добавленные компоненты в соответствии с рис. 33.2.

В класс `Form1` добавьте описания двух полей

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

и вспомогательный метод `Info` (листинг 33.10).

В метод `Form1_Load` добавьте три оператора:

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

В метод `label_Move` добавьте два оператора:

```
count++;
Info();
```

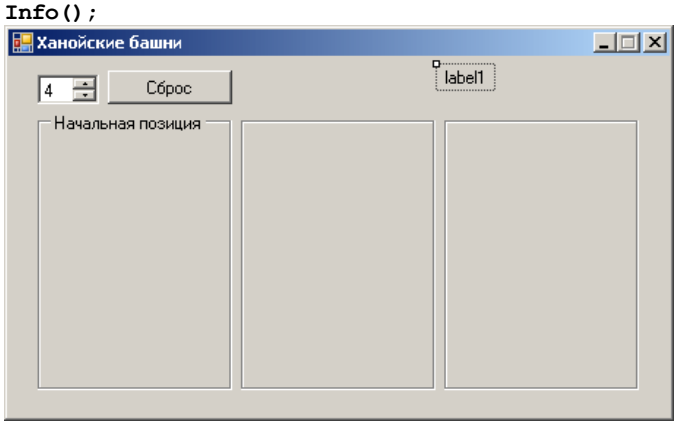

Рис. 33.2. Вид формы Form1 для проекта HTOWERS на промежуточном этапе разработки

Листинг 33.10. Метод `Info` формы `Form1`

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

Результат: для восстановления начальной позиции с тем же количеством блоков следует нажать кнопку **Начальная позиция**. Если при восстановлении начальной позиции требуется изменить число блоков, то по-прежнему достаточно указать новое значение в компоненте `numericUpDown1` (нажимать кнопку в этом случае не требуется).

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 $2^n - 1$).

Примечание

Обратите внимание на то, что перемещения блока в пределах одного и того же компонента `GroupBox` при подсчете числа перемещений не учитываются.

33.5. Контроль за решением задачи

Разместите в форме `Form1` под имеющейся меткой `label1` еще одну метку (`label2`), ее свойству `Text` присвойте значение **`Задача решена!`**, а свойству `ForeColor` — значение **`Green`**.

В метод `Form1_Load` добавьте оператор

```
label2.Visible = false;
```

В метод `label_Move` добавьте следующий фрагмент:

```
if (groupBox2.Controls.Count == numericUpDown1.Value
  || groupBox3.Controls.Count == numericUpDown1.Value)
  label2.Visible = true;
```

Результат: задача считается решенной и об этом выводится сообщение **Задача решена!**, если размер башни в конечной позиции (на компоненте `groupBox2` или `groupBox3`) равен общему числу блоков.

Недочет: после решения задачи по-прежнему разрешено перемещение блоков. **Исправление**: добавьте в начало метода `groupBox1_DragEnter` следующий фрагмент:

```
if (label2.Visible)
{
  e.Effect = DragDropEffects.None;
  return;
}
```

Результат: после решения задачи блоки нельзя перемещать.

33.6. Выполнение задачи в демо-режиме

Разместите в форме `Form1` еще одну кнопку (`button2`) и присвойте ее свойству `Text` значение **`Демо`**. Форма Form1 примет вид, приведенный на рис. 33.3.

В класс `Form1` добавьте новый метод `Step` (листинг 33.11) и определите обработчик события `Click` для кнопки `button2` (листинг 33.12).

В начало метода `label_MouseDown` добавьте следующий фрагмент:

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

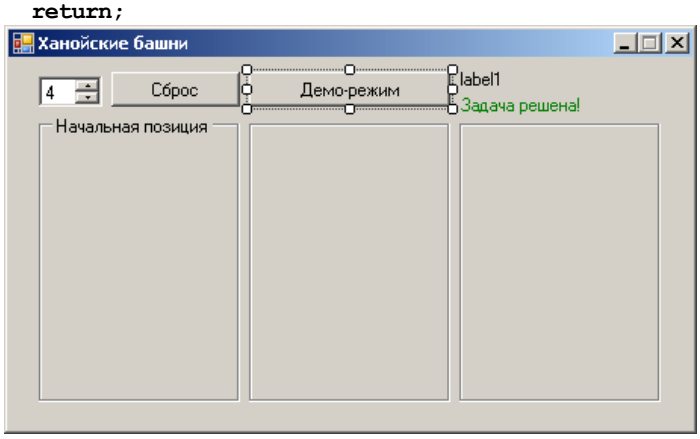

Рис. 33.3. Окончательный вид формы Form1 для проекта HTOWERS

Листинг 33.11. Метод `Step` формы `Form1`

```
private void Step(int n, GroupBox src, GroupBox dst, GroupBox tmp)
{
  if (n == 0)
    return;
  Step(n - 1, src, tmp, dst);
  if (button1.Enabled)
    return;
  label_Move(src.Controls[src.Controls.Count - 1] as Label, dst);
  Application.DoEvents();
  System.Threading.Thread.Sleep(1500 / ((int)numericUpDown1.Value) - 1);
  Step(n - 1, tmp, dst, src);
}
```

Листинг 33.12. Обработчик `button2.Click`

```
private void button2_Click(object sender, EventArgs e)
{
  numericUpDown1.Enabled = button1.Enabled = !button1.Enabled;
  if (!button1.Enabled)
  {
    if (groupBox1.Controls.Count != numericUpDown1.Value)
      numericUpDown1_ValueChanged(null, null);
    Step((int)numericUpDown1.Value, groupBox1, groupBox3, groupBox2);
    numericUpDown1.Enabled = button1.Enabled = true;
  }
}
```

Результат: при нажатии на кнопку **Демо** программа переходит в демо-режим, демонстрирующий правильное решение задачи с указанным числом блоков (блоки перемещаются автоматически). В демо-режиме блокируется компонент `numericUpDown1` и кнопка **Начальная позиция**; кроме того, в нем запрещено перетаскивание меток. 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 ошибка времени выполнения `ArgumentOutOfRangeException`, связанная с тем, что ранее вызванные методы `Step` продолжают выполняться уже после

того, как свойства-коллекции `Controls` компонентов `GroupBox` были очищены в результате завершающих действий программы.

Исправление: определите обработчик события `FormClosed` для формы `Form1` (листинг 33.13).

Листинг 33.12. Обработчик `Form1.FormClosed`

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
  button1.Enabled = true;
}
```

Результат: теперь при закрытии формы кнопка `button1` делается доступной, что позволяет практически немедленно завершить все рекурсивные вызовы метода `Step` без обращения к коллекциям `Controls`.

Приведем изображение работающей программы после решения задачи с пятью блоками (рис. 33.4).

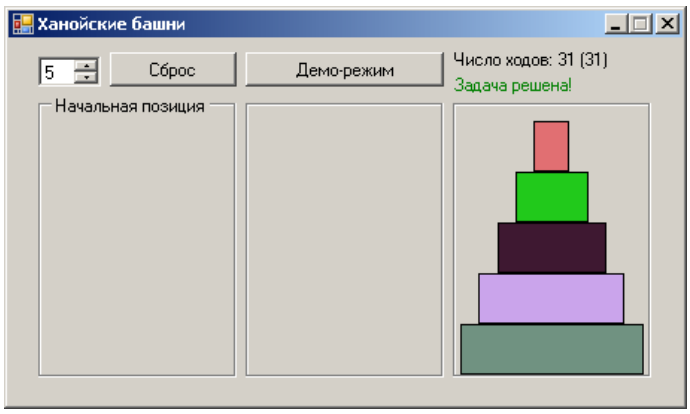

Рис. 33.4. Вид работающего приложения HTOWERS