ГЛАВА 9

Кисти, преобразования и растровые изображения

В предыдущей главе мы начали рассмотрение встроенной в Silverlight двухмерной модели рисования. Было рассмотрено использование классов, производных от класса `Shape`, таких как `Rectangle`, `Ellipse`, `Polygon`, `Polyline` и `Path`, для создания фигур. Однако для создания двухмерных векторных рисунков для мощных приложений Silverlight фигур недостаточно. В данной главе будут представлены дополнительные средства рисования.

В первую очередь мы обсудим экзотические кисти Silverlight, позволяющие создавать градиенты, перекрывающиеся шаблоны и растровые изображения, которыми можно заполнять фигуры. Затем будет рассмотрено использование средств управления прозрачностью для объединения изображений и элементов. И наконец, мы представим объекты преобразований, изменяющие внешний вид любого элемента путем его масштабирования, поворота или наклона. С помощью перечисленных средств можно создавать разнообразные визуальные эффекты, включая отражение, сияние и тени.

Новые средства. В конце данной главы будут описаны три новых средства Silverlight. Наиболее впечатляющее — перспективные преобразования, позволяющие имитировать трехмерные эффекты. Затем рассматривается раскрашивание пикселей — технология наложения сложных визуальных эффектов на любой элемент. И наконец, будет представлен класс `WriteableBitmap`, предназначенный для изменения отдельных пикселей растрового изображения, даже когда оно уже отображено на экране..

Кисти

Кисть заполняет заданную область, в качестве которой может выступать фон, передний план, рамка элемента, штрих или фигура. Для элементов Silverlight кисти задаются с помощью свойств `Foreground`, `Background` и `BorderBrush`. Для фигур кисти задаются в свойствах `Fill` и `Stroke`.

В предыдущих главах кисти применялись неоднократно, однако в большинстве случаев использовалась наиболее простая кисть `SolidColorBrush`. Без сомнения, она весьма полезна, однако существует ряд более мощных классов, производных от `System.Windows.Media.Brush` и позволяющих создавать экзотические эффекты. Классы кистей перечислены в табл. 9.1.

Таблица 9.1. Классы кистей

Имя кисти	*Описание*
SolidColorBrush	Заполнение области одним цветом
LinearGradientBrush	Заполнение области линейным градиентом; оттенок плавно переходит от одного цвета к другому (затем, возможно, к третьему, четвертому и т.д.)
RadialGradientBrush	Заполнение области радиальным градиентом; от линейного он отличается тем, что цвет изменяется вдоль радиуса
ImageBrush	Заполнение области растровым изображением, которое может растягиваться, масштабироваться, перекрываться и т.д.
VideoBrush	Заполнение области медийным элементом MediaElement, воспроизводящим видеофайл; таким образом, видеофайл можно воспроизводить в любой фигуре или любом элементе

В данной главе рассматривается использование кистей linearGradientBrush, RadialGradientBrush и ImageBrush. Кисть VideoBrush рассматривается в главе 11 при обсуждении медийных средств.

Кисть LinearGradientBrush

Эта кисть позволяет создавать заливку, цвет которой плавно изменяется по линейному закону.

Ниже приведена разметка прямоугольника, заполненного простейшим линейным градиентом. Цвет заливки изменяется от синего в левом верхнем углу до белого в правом нижнем углу.

```
<Rectangle Width="150" Height="100">
  <Rectangle.Fill>
    <LinearGradientBrush >
      <GradientStop Color="Blue" Offset="0"/>
      <GradientStop Color="White" Offset="1" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

Результат показан в верхней части рис. 9.1.

Для создания первого градиента нужно добавить по одному элементу GradientStop для каждого цвета. Цвет размещается в градиенте с помощью свойства Offset (Смещение), которое может принимать значение от 0 до 1. В данном примере элемент GradientStop для синего цвета имеет смещение 0. Это означает, что синий цвет размещается в самом начале градиента. Для белого цвета смещение равно 1, следовательно, белый цвет размещается в самом конце градиента. Значения Offset определяют, как быстро цвет изменяется от одного к другому. Например, если для белого цвета установить смещение 0.5, градиент начнется с синего цвета в левом верхнем углу и перейдет в белый цвет посредине пути (в точке между двумя противоположными углами прямоугольника). Правая нижняя часть прямоугольника будет полностью белой (второй сверху прямоугольник на рис. 9.1).

Приведенная выше разметка создает градиент с диагональным заполнением, изменяющийся от одного угла прямоугольника до противоположного. С помощью свойств StartPoint и EndPoint кисти LinearGradientBrush можно создавать градиенты, изменяющиеся в любом заданном направлении. Эти свойства задают точку, в которой

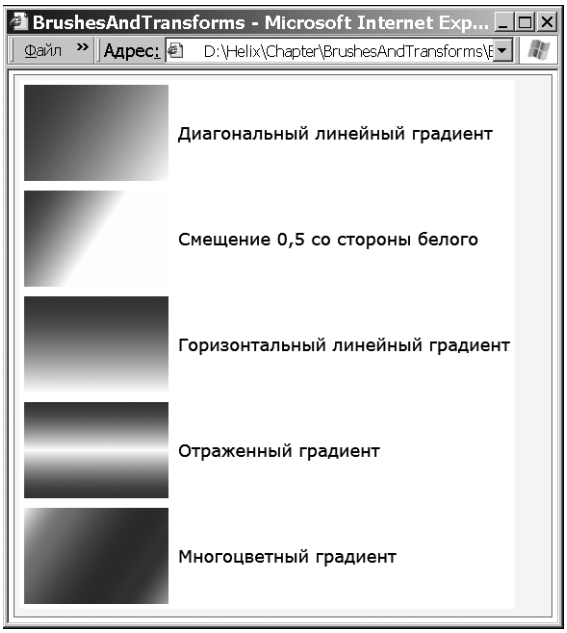

Рис. 9.1. Прямоугольники, заполненные разными линейными градиентами

начинает изменяться первый цвет, и точку, в которой заканчивает изменяться последний цвет. Необходимо учитывать, что свойства `StartPoint` и `EndPoint` содержат не абсолютные, а относительные координаты точек. Объект `LinearGradientBrush` присваивает координаты `0,0` левому верхнему углу и координаты `1,1` — правому нижнему углу заполняемой области независимо от ее высоты или ширины.

Чтобы создать горизонтальный градиент (третий сверху на рис. 9.1), необходимо установить начальную точку в левый верхний угол `0,0`, а конечную точку — в левый нижний угол `0,1`. Аналогично, чтобы создать вертикальный градиент, начальная точка должна иметь координаты `0,0`, а конечная — `1,0`.

Начальная и конечная точки не обязательно должны совпадать с углами заполняемой области. Например, если присвоить начальной точке координаты `0,0`, а конечной — `0,0.5`, будет создан сжатый линейный градиент: первый цвет начинается на самом верху области, а второй заканчивается посредине области. Нижняя часть области заполняется вторым цветом. Изменить такое поведение градиента можно с помощью свойства `LinearGradientBrush.SpreadMethod`. При установленном по умолчанию значении `Pad` область за пределами градиента заполняется ближайшим цветом. При значении `Reflect` градиент "переворачивается", т.е. сначала первый цвет вытесняется вторым, а затем второй — первым (четвертый сверху градиент на рис. 9.1). При значении `Repeat` заданная последовательность цветов повторяется.

Градиент может содержать более двух цветов. Для этого в кисть нужно добавить более двух объектов `GradientStop`.

```
<Rectangle Width="150" Height="100">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="Red" Offset="0.25" />
```

```
      <GradientStop Color="Blue" Offset="0.75" />
      <GradientStop Color="LimeGreen" Offset="1.0" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

Для каждого объекта `GradientStop` нужно задать смещение `Offset`. Если смещение наращивается равномерно, полоски имеют одну и ту же ширину. Но можно задать, например, быстрое изменение градиента в начале и медленное в конце, например, если установить ряд смещений `0`, `0.1`, `0.2`, `0.4`, `0.6` и `1`.

Не забывайте, что кисти можно применять не только в фигурах. Кисть `LinearGradientBrush` можно вставить в любом месте разметки, в котором разрешен объект `SolidColorBrush`, например в свойство `Background` для заполнения фона любого элемента; в свойство `Foreground` — для заполнения цветом текста (рис. 9.2); или в свойство `BorderBrush` — для заполнения рамки (не области, ограниченной рамкой, а самой рамки).

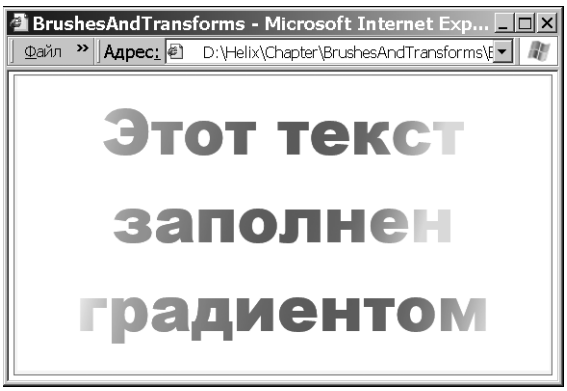

Рис. 9.2. Присвоение кисти `LinearGradientBrush` свойству `TextBlock.Foreground`

Кисть `RadialGradientBrush`

Объект `RadialGradientBrush` работает аналогично объекту `LinearGradientBrush`. Он тоже принимает последовательность цветов с разными смещениями. Как и в `LinearGradientBrush`, в нем можно задать произвольное количество цветов. Отличие состоит лишь в способе изменения цветов.

Для определения точки, в которой начинается первый цвет градиента, используется свойство `GradientOrigin`. По умолчанию оно равно `0.5,0.5`, в результате чего первый цвет начинается в середине заполняемой области.

Примечание. Как и в `LinearGradientBrush`, в кисти `RadialGradientBrush` используется пропорциональная система координат. Верхний левый угол прямоугольной области заполнения имеет координаты `0,0`, а нижний правый — координаты `1,1`. Это означает, что если в качестве начальной точки градиента выбрать точку между `0,0` и `1,1`, заполнение начнется внутри прямоугольника. Можно также задать начальную точку за пределами этих границ, тогда заполнение начнется снаружи.

Цвет изменяется от начальной точки вдоль радиусов и заканчивает изменяться на внутренней окружности градиента. Центр внутренней окружности может не совпадать с начальной точкой градиента. Область за пределами внутренней окруж-

ности заполняется последним цветом, заданным в коллекции `RadialGradientBrush.GradientStops` (рис. 9.3). Градиент может содержать произвольное количество цветов. Каждый цвет определяется одним объектом `GradientStop`.

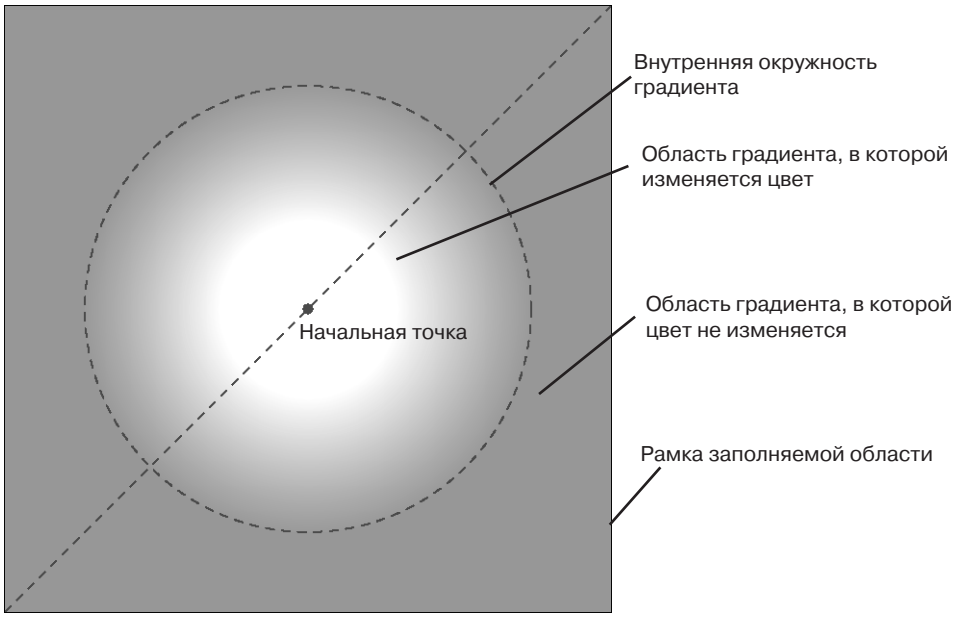

Рис. 9.3. Радиальный градиент

Внутренняя окружность градиента определяется свойствами `Center`, `RadiusX` и `RadiusY`. По умолчанию свойство `Center` равно `0.5,0.5`, и внутренняя окружность помещается в середине заполняемой области, причем ее центр совпадает с начальной точкой градиента.

Значения `RadiusX` и `RadiusY` по умолчанию равны `0.5`. Радиусы вычисляются относительно половины диагонали заполняемой области. Это означает, что при значениях `0.5` радиусы равны четверти диагонали. Если заполняемая область квадратная и значения равны `0.5`, то, согласно теореме Пифагора, диаметр внутренней окружности будет равен 0,7 ширины или высоты заполняемой области.

Примечание. Если радиусы не равны, внутренняя окружность фактически является эллипсом.. Объекты `RadialGradientBrush` обычно используются для заполнения круглых фигур и создания эффектов освещенности. Например, с их помощью можно создавать кнопки, которые выглядят так, будто они светятся. Эффект глубины создается путем смещения начальной точки с помощью свойства `GradientOrigin`. Ниже приведена разметка смещенного градиента.

```
<Ellipse Margin="5" Stroke="Black" StrokeThickness="1"
 Width="200" Height="200">
  <Ellipse.Fill>
    <RadialGradientBrush RadiusX="1" RadiusY="1"
      GradientOrigin="0.7,0.3">
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="Blue" Offset="1" />
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Результат показан на рис. 9.4 наряду с несмещенным градиентом, в котором свойство `GradientOrigin` равно `0.5,0.5`.

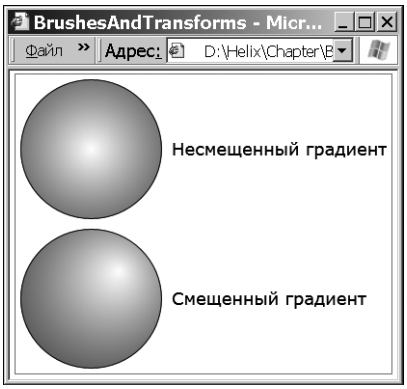

Рис. 9.4. Смещение радиального градиента

Кисть `ImageBrush`

С помощью кисти `ImageBrush` область можно заполнять растровым изображением, хранящимся в файле допустимого типа. Надстройка Silverlight поддерживает файлы BMP, PNG и JPEG. Файл идентифицируется свойством `ImageSource`. Например, приведенная ниже кисть заполняет фон контейнера `Grid` изображением `logo.jpg`, включенным в проект в качестве ресурса (и, следовательно, внедренным в файл приложения XAP).

```
<Grid>
  <Grid.Background>
    <ImageBrush ImageSource="logo.jpg"></ImageBrush>
  </Grid.Background>
</Grid>
```

Свойство `ImageSource` объекта `ImageBrush` работает так же, как свойство `Source` элемента `Image`. Это означает, что хранящийся в нем адрес URI может указывать на файл, внедренный в проект или размещенный в Интернете.

Примечание. Надстройка Silverlight учитывает информацию об изображении, хранящуюся в файле вместе с изображением. Например, в Silverlight поддерживаются полупрозрачные области, определенные в файлах PNG.

В приведенной выше разметке кисть `ImageBrush` заполняет фон ячейки. Поэтому по умолчанию изображение растягивается, пока не заполнит всю область. Если ячейка `Grid` больше исходного изображения, могут проявиться эффекты растягивания, например искажение пропорций или пушистость. Управлять поведением кисти можно с помощью свойства `ImageBrush.Stretch` (табл. 9.2).

Если изображение меньше заполняемой области, оно выравнивается соответственно значениям свойств `AlignmentX` и `AlignmentY`. Незаполненная часть остается пустой. Это происходит также при значении `Uniform` и разных пропорциях изображения и заполняемой области. Кроме того, пустое пространство может остаться при значении `None`, когда изображение меньше заполняемой области.

Таблица 9.2. Элементы перечисления `Stretch`

Имя элемента	*Описание*
`Fill`	Изображение растягивается по ширине и высоте, пока не заполнит весь контейнер; пропорции не сохраняются; значение `Fill` установлено по умолчанию
`None`	Изображение не растягивается; устанавливаются размеры изображения, определенные в файле; не поместившаяся часть изображения отсекается; если изображение меньше контейнера, в контейнере остается пустое пространство
`Uniform`	Ширина и высота изображения увеличиваются пропорционально, пока не будет достигнута одна из границ контейнера; пропорции сохраняются, но может остаться пустое пространство
`UniformToFill`	Ширина и высота изображения увеличиваются пропорционально, пока не будет заполнено все доступное пространство; пропорции сохраняются; не поместившаяся часть изображения отсекается

Прозрачность

В примерах, обсуждавшихся до сих пор, все фигуры были непрозрачными. Однако Silverlight поддерживает частичную прозрачность. Это означает, что если вывести разные элементы один поверх другого и присвоить им разные степени прозрачности, нижележащие элементы будут видны сквозь вышележащие. Это позволяет создавать различные визуальные эффекты, в том числе анимированные.

Существует несколько способов сделать элемент полупрозрачным.

- **Установка свойства `Opacity` элемента.** Оно может принимать значения от `0` до `1`. При значении `1` (установлено по умолчанию) элемент непрозрачный, а при значении `0` — полностью прозрачный. Свойство `Opacity` определено в классе `UIElement`, поэтому оно присуще всем элементам.
- **Установка свойства `Opacity` кисти.** Как и элементы, классы кистей содержат свойство `Opacity`, позволяющее сделать заливку полупрозрачной. Кисть можно использовать для заполнения части элемента.
- **Использование полупрозрачных цветов.** Любой цвет со значением `alpha`, меньшим `255`, выводится полупрозрачным. Полупрозрачные цвета можно использовать для заполнения фона, переднего плана или рамки элемента.
- **Установка свойства `OpacityMask`.** Маска позволяет сделать заданную область элемента прозрачной или полупрозрачной. Например, ее можно использовать для создания плавного перехода от прозрачной области к непрозрачной.

На рис. 9.5 продемонстрировано использование первых двух способов создания полупрозрачных элементов.

В данном примере фон контейнера `Grid` верхнего уровня заполнен растровым изображением с помощью кисти `ImageBrush`. Свойство `Opacity` контейнера `StackPanel` равно `0.7`, в результате чего просматривается нижележащий цвет (в данном случае он белый, и рисунок всего лишь немного осветляется).

```
<Grid Margin="5" Opacity="0.7">
  <Grid.Background>
    <ImageBrush ImageSource="celestial.jpg" />
  </Grid.Background>
  ...
</Grid>
```

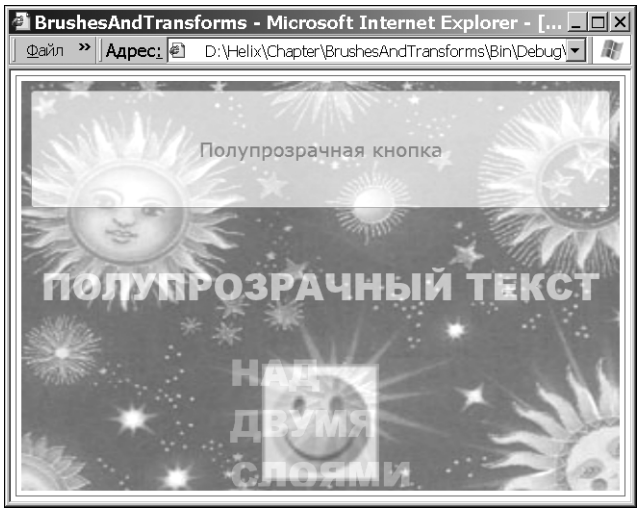

Рис. 9.5. Страница, содержащая полупрозрачные элементы

В кнопке используется полупрозрачный красный цвет фона, установленный с помощью свойства `Background`. Изображение видно сквозь кнопку, однако текст кнопки непрозрачный (если бы свойство `Opacity` было установлено и для фона, и для переднего плана, текст кнопки тоже был бы полупрозрачным). Ниже приведена разметка кнопки.

```
<Button Foreground="Green" Background="#60AA4030"
 FontSize="16" Margin="10"
Padding="20" Content="Полупрозрачная кнопка"></Button>
```

Примечание. В Silverlight применяется цветовой стандарт ARGB, в котором для описания каждого цвета используется четыре значения. Каждое значение должно быть целым числом в диапазоне от 0 до 255. Первое значение (`alpha`) определяет степень прозрачности. При значении `0` цвет полностью прозрачный, а при значении `1` — непрозрачный. Значения RGB определяют интенсивность красного, зеленого и синего компонентов цвета.

Следующий элемент — `TextBlock`. По умолчанию все элементы `TextBlock` имеют полностью прозрачный фон. В данном примере прозрачность фона оставлена установленной по умолчанию, но свойство `Opacity` делает текст полупрозрачным. Этого же эффекта можно достичь, установив для свойства `Foreground` белый цвет с ненулевым значением `alpha`.

```
<TextBlock Grid.Row="1" Margin="10" TextWrapping="Wrap"
 Foreground="White" Opacity="0.3" FontSize="38"
 FontFamily="Arial Black"
Text="ПОЛУПРОЗРАЧНЫЙ ТЕКСТ"></TextBlock>
```

Последним размещен вложенный элемент `Grid`, который содержит два элемента в одной ячейке, один поверх другого (для перекрытия двух элементов можно также использовать элемент `Canvas`, позволяющий точно позиционировать содержимое). Внизу размещен полупрозрачный элемент `Image` с изображением шутливой мордочки. В нем тоже установлено свойство `Opacity`, позволяющее видеть сквозь него нижележащее изображение. Поверх обоих изображений размещен элемент `TextBlock` с полупрозрачным текстом. Сквозь буквы видны оба изображения. Ниже приведена разметка изображения.

```
<Image Grid.Row="2" Margin="10" Source="happyface.jpg"
 Opacity="0.5" ></Image>
```

Можно накладывать одно поверх другого произвольное количество изображений и элементов, сделав их полупрозрачными. Конечно, при этом ухудшается быстродействие прорисовки, особенно если в приложении применяются динамические эффекты, такие как анимация. Кроме того, если полупрозрачных слоев более трех, сквозь них становится тяжело что-либо увидеть.

Маска

Свойство `OpacityMask` делает заданную область элемента прозрачной или полупрозрачной, позволяя таким образом создавать экзотические эффекты. Например, фигуру можно сделать плавно переходящей от полностью прозрачного до непрозрачного состояния.

Свойство `OpacityMask` может принимать любую кисть. Значение `alpha` кисти определяет степень прозрачности. Например, если для маски используется кисть `SolidColorBrush` с полностью прозрачным цветом, элемент будет невидим. Если же в кисти задан непрозрачный цвет, элемент останется видимым. Другие параметры цвета (интенсивности красного, зеленого и синего компонентов) при установке свойства `OpacityMask` игнорируются.

Использовать маску с кистью `SolidColorBrush` нерационально, потому что того же эффекта легче достичь с помощью свойства `Opacity`. Маска `OpacityMask` полезна с другими типами кистей, такими как `LinearGradientBrush` или `RadialGradientBrush`. С помощью градиентов, переходящих от прозрачного к непрозрачному цвету, можно создать эффект плавной прорисовки элемента.

```
<Button FontSize="14" FontWeight="Bold"
 Content="Частично прозрачная кнопка">
  <Button.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
      <GradientStop Offset="0" Color="Transparent">
      </GradientStop>
      <GradientStop Offset="0.8" Color="Black">
      </GradientStop>
    </LinearGradientBrush>
  </Button.OpacityMask>
</Button>
```

Результат показан на рис. 9.6. Сквозь кнопку плавно прорисовывается пианино.

Рис. 9.6. Кнопка с маской

Полупрозрачные элементы управления

До сих пор мы рассматривали создание полупрозрачных элементов, сквозь которые видны другие элементы Silverlight. Однако возможен и другой подход: область содержимого Silverlight можно вывести без окна, в результате чего сквозь элементы Silverlight будет видно содержимое HTML.

Конфигурирование Silverlight на вывод содержимого без окна состоит из нескольких этапов. В первую очередь нужно отредактировать разметку XAML, удалив из нее непрозрачный фон. Обычно при создании новой страницы в программе Visual Studio она добавляет в разметку единственный контейнер `Grid`, заполняющий всю страницу. Контейнер `Grid` является корневым элементом страницы, поэтому Visual Studio явно присваивает ему белый непрозрачный фон.

```
<Grid x:Name="LayoutRoot" Background="White">
```

Чтобы страница была прозрачной, нужно удалить свойство `Background`. Тогда контейнер `Grid` получит прозрачный фон, установленный для него по умолчанию.

Затем нужно отредактировать входную страницу HTML. Найдите элемент `<div>`, в котором размещена область содержимого Silverlight. Измените значение параметра `background` с `white` на `transparent` и добавьте параметр `windowless`, присвоив ему значение `true`.

```
<div id="silverlightControlHost">
  <object data="data:application/x-silverlight,"
   type="application/x-silverlight-2" width="100%"
   height="100%">
    <param name="source"
     value="TransparentSilverlight.xap"/>
    <param name="onerror" value="onSilverlightError" />
    <param name="background" value="transparent" />
    <param name="windowless" value="true" />
    ...
  </object>
  <iframe
   style='visibility:hidden;height:0;width:0;border:0px'>
  </iframe>
</div>
```

На рис. 9.7 и 9.8 показаны примеры размещения области содержимого Silverlight на странице HTML. На рис. 9.7 элемент Silverlight выводится по умолчанию с непрозрачным фоном, а на рис. 9.8 этот же элемент Silverlight выводится в безоконном режиме с прозрачным фоном, сквозь который видно содержимое HTML.

Безоконный режим Silverlight позволяет не только увидеть содержимое HTML сквозь содержимое Silverlight, но и вывести содержимое HTML поверх содержимого Silverlight. На рис. 9.8 этот эффект проявляется в том, что фраза `Это текст HTML` не затеняется содержимым Silverlight.

Для создания данного примера два элемента `<div>` позиционированы на основе абсолютных координат в левой части страницы с помощью двух классов.

```
.SilverlightLeftPanel
{
  background-image: url('tiles5x5.png');
  background-repeat:repeat;
  position: absolute;
  top: 70px;
  left: 10px;
```

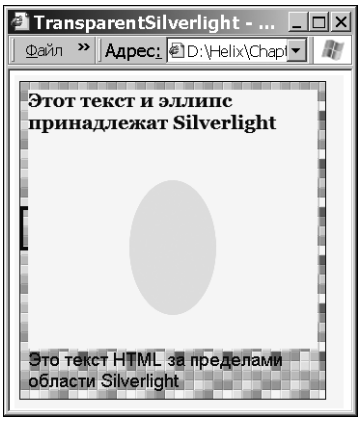

Рис. 9.7. Обычный режим вывода содержимого Silverlight

Рис. 9.8. Безоконная область содержимого Silverlight

```
  width: 142px;
  border-width: 1px;
  border-style: solid;
  border-color: black;
  padding: 8px;
}

.HtmlLeftPanel
{
  background-color: Transparent;
  position: absolute;
  top: 300px;
  left: 10px;
  width: 142px;
  font-weight: bold;
  border-width: 1px;
  border-style: solid;
  border-color: black;
  padding: 8px;
}
```

В первом элементе `<div>` размещена область содержимого Silverlight, а во втором — перекрывающееся содержимое HTML.

```
<div class="SilverlightLeftPanel">
  <div id="silverlightControlHost">...</div>
</div>

<div class="HtmlLeftPanel" >
  <p>Это текст HTML</p>
</div>
```

Совет. Обычно безоконные элементы Silverlight используются для бесшовного совмещения непрямоугольного содержимого Silverlight с фоном нижележащей страницы HTML. Однако их можно использовать и в других целях. Например, для размещения элементов HTML и Silverlight рядом. Это особенно полезно, если элементы взаимодействуют (см. главу 14). В частности, при создании проигрывателя Silverlight с кнопками HTML рекомендуется использовать безоконный элемент Silverlight.. Использование безоконных областей содержимого Silverlight приводит

к дополнительным затратам времени и оперативной памяти (заметным при частой перерисовке изображения или применении анимации), поэтому добавляйте их в приложение только при необходимости. При использовании безоконной области содержимого не предполагайте, что элементу Silverlight автоматически присваивается белый фон. При выполнении приложения на компьютере Mac всегда используется безоконный режим, независимо от передаваемых параметров. Поэтому на входной странице должен быть явно установлен белый фон.

Объекты преобразований

Многие задачи рисования можно существенно упростить с помощью объектов преобразований, которые изменяют внешний вид фигур и элементов путем изменения используемой системы координат. В Silverlight объекты преобразований представлены классами, производными от абстрактного класса `System.Windows.Media.Transform` (табл. 9.3).

Таблица 9.3. Классы преобразований

Имя класса	*Описание*	*Важные свойства*
`TranslateTransform`	Смещение системы координат; этот объект полезен, когда нужно нарисовать одну и ту же фигуру в разных местах	`X`, `Y`
`RotateTransform`	Поворот системы координат; существующая фигура поворачивается вокруг заданной центральной точки	`Angle`, `CenterX`, `CenterY`
`ScaleTransform`	Масштабирование системы координат; фигуры становятся больше или меньше; по осям X и Y можно задать разные коэффициенты масштабирования, искажая таким образом пропорции фигуры; при изменении размеров фигуры пропорционально изменяются также размеры внутренней области и рамки; чем больше становится фигура, тем толще рамка после преобразования; объект `ScaleTransform` довольно похож на объект `Viewbox` (см. главу 8)	`ScaleX`, `ScaleY`, `CenterX`, `CenterY`
`SkewTransform`	Наклон системы координат; например, в результате наклона прямоугольник превращается в параллелограмм	`AngleX`, `AngleY`, `CenterX`, `CenterY`
`MatrixTransform`	Изменение системы координат путем умножения на заданную матрицу; это довольно сложное преобразование, требующее знания математики	Матрица
`TransformGroup`	Объединение многих преобразований для их выполнения за один раз; важна последовательность преобразований, потому что она влияет на результат; например, если сначала повернуть (`RotateTransform`), а затем переместить (`TranslateTransform`) фигуру, она будет перемещена в некотором направлении, однако если выполнить эти операции в обратной последовательности, фигура будет перемещена в совершенно другом направлении	Нет

Технически во всех преобразованиях для изменения координат фигуры используется матричная математика. Однако использование встроенных объектов преобразований `TranslateTransform`, `RotateTransform`, `ScaleTransform` и `SkewTransform` позволяет изменять фигуры без явного применения сложных формул. Они остаются "за кадром". Желающие применить свои знания математики могут использовать объект `MatrixTransform`. Кроме того, при задании ряда преобразований с помощью объекта `TransformGroup` надстройка Silverlight автоматически создает результирующую формулу и оптимизирует ее реализацию.

Примечание. Все преобразования поддерживают автоматическое извещение об изменениях. Это означает, что при программном изменении объекта преобразований, используемого в фигуре, она автоматически обновляется..

Объекты преобразований полезны при решении многих задач, включая следующие.

- **Создание повернутых фигур.** С помощью объекта `RotateTransform` легко создавать повернутые фигуры.
- **Копирование фигур.** На некоторых рисунках одинаковые фигуры размещаются в разных местах. Используя объекты преобразований, можно создать одну фигуру, а затем разместить на странице ее копии, причем в другом виде, например повернутые, с измененными размерами и т.д.

Совет. Существует три способа использования одной и той же фигуры в разных местах: дублирование определения фигуры в разметке (не идеальный вариант), программное копирование фигуры в коде и копирование фигуры с помощью объекта `Path` (см. главу 8). Элемент `Path` принимает объект `Geometry`, поэтому геометрический объект можно сохранить как ресурс и применять его в разных местах разметки..

- **Динамические эффекты и анимация.** С помощью объектов преобразований можно создавать довольно изощренные эффекты, такие как плавный поворот или перемещение фигуры.

В главе 10 рассматривается использование преобразований в анимации. Однако пока рассмотрим применение преобразований к обычным фигурам.

Преобразование фигур

Для преобразования фигуры нужно включить соответствующий объект преобразования в свойство `RenderTransform` фигуры. В зависимости от используемого объекта преобразования, для его конфигурирования нужно заполнить свойства, описанные выше в табл. 9.3.

Рассмотрим поворот прямоугольника с помощью объекта `RotateTransform`. Естественно, нужно задать угол поворота. Приведенная ниже разметка создает прямоугольник, повернутый на угол 25°.

```
<Rectangle Width="80" Height="10" Stroke="Blue"
 Fill="Yellow" Canvas.Left="100" Canvas.Top="100">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" />
  </Rectangle.RenderTransform>
</Rectangle>
```

По умолчанию фигура поворачивается вокруг своего начала координат, т.е. вокруг верхнего левого угла занимаемой области. На рис. 9.9 показан прямоугольник, повернутый на углы 29, 50, 75 и 100°.

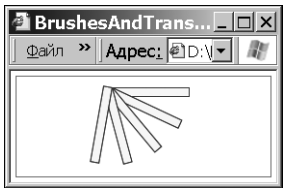

Рис. 9.9. Поворот прямоугольника

Иногда фигуру нужно повернуть вокруг точки, не совпадающей с началом координат. Для этого объект `RotateTransform`, как и другие классы преобразований, предоставляет свойства `CenterX` и `CenterY`, используемые для задания центра поворота. В приведенной ниже разметке прямоугольник поворачивается на 25° вокруг точки, расположенной рядом с его центром.

```
<Rectangle Width="80" Height="10" Stroke="Blue"
 Fill="Yellow" Canvas.Left="100" Canvas.Top="100">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" CenterX="45" CenterY="5" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Результат показан на рис. 9.10 (наряду с поворотами на другие углы).

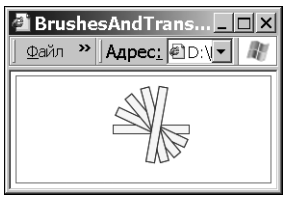

Рис. 9.10. Поворот прямоугольника вокруг заданной точки

Значения свойств `CenterX` и `CenterY` задаются в абсолютных координатах. Это означает, что для задания центра поворота нужно знать начало координат содержимого. Однако если выводится динамическое содержимое (например, элементы или рисунки с изменяемыми размерами), возникают проблемы. Для их решения можно воспользоваться удобным свойством `RenderTransformOrigin`, поддерживаемым всеми фигурами. Оно устанавливает центральную точку на основе пропорциональной системы координат в диапазоне от 0 до 1 в обоих измерениях. Точка `0,0` находится в левом верхнем углу области содержимого, а точка `1,1` — в правом нижнем углу. Если область содержимого не квадратная, система координат растягивается соответствующим образом.

Приведенная ниже разметка с помощью свойства `RenderTransformOrigin` поворачивает прямоугольник вокруг его центральной точки на угол 25°.

```
<Rectangle Width="80" Height="10" Stroke="Blue"
 Fill="Yellow" Canvas.Left="100" Canvas.Top="100"
 RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Точка `0.5,0.5` обозначает центр фигуры независимо от ее размеров. На практике свойство `RenderTransformOrigin` полезнее и применяется чаще, чем свойства `CenterX` и `CenterY`, однако в каждой конкретной ситуации нужно выбрать оптимальное свойство на основе специфики решаемой задачи.

Совет. В свойстве `RenderTransformOrigin` можно использовать значения больше `1` и меньше `0`. Тогда центр преобразования будет установлен за пределами области содержимого фигуры. Таким способом можно повернуть фигуру по большой дуге, задав точку, удаленную на большое расстояние, например `5,5`.

Преобразования и контейнеры

Свойства `RenderTransform` и `RenderTransformOrigin` поддерживаются не только фигурами, производными от класса `Shape`. Эти свойства наследуются от класса `UIElement`, поэтому их можно использовать в любых элементах Silverlight, включая кнопки, текстовые блоки, контейнеры и т.д. Поворачивать, наклонять и масштабировать можно любые компоненты пользовательского интерфейса Silverlight (хотя в большинстве случаев делать этого не нужно).

Рис. 9.11. Поворот кнопки

Важно учитывать, что преобразование элементов, находящихся в контейнере, выполняется контейнером после их размещения. В простом контейнере `Canvas`, в котором элементы размещаются на основе заданных координат, последовательность операций (сначала размещение, а потом преобразование или наоборот) не играет роли. Однако в других контейнерах, в которых размещение выполняется относительно соседних элементов и зависит от их размеров, последовательность операций важна. Рассмотрим пример, показанный на рис. 9.11. Контейнер `StackPanel` содержит повернутую кнопку. Кнопки сначала размещаются, а потом поворачиваются. Поэтому вторая и четвертая кнопки размещены таким образом, будто третья кнопка не поворачивалась. В результате кнопки перекрываются. Если бы размещение выполнялось после поворота, кнопки не перекрывались бы.

В WPF можно задать размещение после преобразования. При этом результат преобразования учитывается при позиционировании соседних элементов. Однако в Silverlight такая возможность не предусмотрена.

Совет. Объекты преобразований можно использовать в любых компонентах Silverlight, например в кистях, объектах `Geometry`, областях отсечения и т.д.

Создание эффекта отражения

Объекты преобразований позволяют создавать эффекты различных типов. Например, с их помощью можно создать эффект зеркального отражения (рис. 9.12).

Для создания в Silverlight эффекта отражения нужно явно продублировать содержимое. Например, при создании рис. 9.12 были использованы два идентичных элемента `Image`: один элемент `Image` содержит исходное изображение, а второй — отраженную копию.

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
```

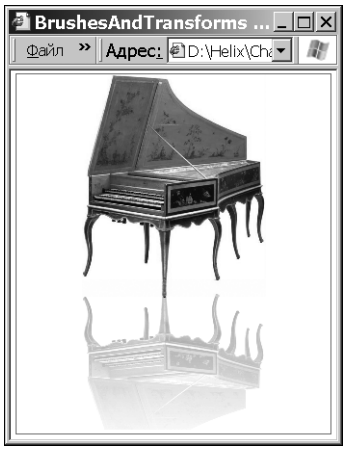

Рис. 9.12. Эффект зеркального отражения

```
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Image Source=" harpsichord.jpg"></Image>
  <Image Grid.Row="1" Source=" harpsichord.jpg"></Image>
</Grid>
```

Необходимость дублировать содержимое приводит к непрактичности данного способа для элементов управления. Кроме того, в главе 11 описан еще один способ создания эффекта отражения при воспроизведении видеофайлов с помощью кисти `VideoBrush`.

Отраженную копию нужно немного изменить, чтобы она выглядела, как отражение. Это можно сделать с помощью двух компонентов: объекта преобразования, размещающего отраженное изображение в нужном месте, и маски, плавно выводящей содержимое копии.

```
<Image Grid.Row="1" Source="harpsichord.jpg"
 RenderTransformOrigin="0,0.4">
  <Image.RenderTransform>
    <ScaleTransform ScaleY="-0.8"></ScaleTransform>
  </Image.RenderTransform>
  <Image.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Offset="0" Color="Transparent">
      </GradientStop>
      <GradientStop Offset="1" Color="#44000000">
      </GradientStop>
    </LinearGradientBrush>
  </Image.OpacityMask>
</Image>
```

В данной разметке объект `ScaleTransform` размещает отраженное изображение в нужном месте с помощью отрицательного значения свойства `ScaleY`. Чтобы точно отразить изображение, нужно присвоить ему значение `-1`. В данном примере свойству `ScaleY` присвоено значение `-0.8`, чтобы изображение не только отразилось, но и немного сжалось для большей реалистичности. Для точного размещения отраженной копии нужно явно позиционировать ее с помощью контейнера `Canvas` или применить свойство `RenderTransformOrigin`, как в данном примере. Изображение отражено относительно точки `0,0.4`. Следовательно, выравнивание влево оставлено прежним (`x=0`), а смещение

равно -0.4. Обратите внимание на то, что отражение выполнено относительно воображаемой горизонтальной линии, расположенной немного выше середины изображения.

Для создания эффекта плавной прорисовки изображения используется кисть LinearGradientBrush. Изображение перевернуто вверх ногами, поэтому объекты GradientStop определены в обратной последовательности.

Перспективные преобразования

В Silverlight нет полнофункционального набора инструментов для трехмерного рисования, но есть средства *перспективных преобразований*, позволяющие имитировать трехмерные поверхности. Как и при обычном преобразовании, объект перспективного преобразования манипулирует внешним видом существующего двухмерного элемента, проецируя его на повернутую поверхность. В результате манипуляций элемент выглядит так, будто он находится на трехмерной поверхности.

Перспективные преобразования весьма полезны, однако им далеко до реальных трехмерных моделей. Их главный, самый очевидный, недостаток состоит в том, что преобразованию подвергается только одна фигура — плоский прямоугольник, на котором размещены элементы. Сравните: в действительно трехмерных моделях для построения сложных поверхностей используются крошечные совмещаемые треугольники. Для получения их координат и освещенности применяются сложные математические формулы, требующие огромного объема вычислений. Мощные средства реального трехмерного моделирования есть в WPF.

Примечание. Если вам нужны всего лишь несколько трюков, чтобы имитировать простые трехмерные эффекты, не затрачивая слишком много труда, вам понравятся средства перспективного преобразования, встроенные в Silverlight. Они особенно полезны в сочетании с анимацией (см. главу 10). Однако если вам необходимы настоящие трехмерные модели, перспективного преобразования вам будет явно недостаточно.

В Silverlight определен абстрактный класс Transform, наследуемый всеми классами двухмерных преобразований. Аналогично этому, все классы проецирования наследуют абстрактный класс System.Windows.Media.Projection. В настоящий момент Silverlight поддерживает два класса проецирования: PlaneProjection, который рассматривается в данной главе, и намного более сложный Matrix3DProjection, в котором преобразования определяются с помощью трехмерных матриц. В данной книге класс Matrix3DProjection не рассматривается. Если хотите поэкспериментировать с ним, загрузите со страниц www.tinyurl.com/m29v3q и www.tinyurl.com/laalp6 примеры кода, созданные Чарльзом Петцольдом (Charles Petzold).

Класс PlaneProjection

Класс PlaneProjection позволяет поворачивать и смещать плоское изображение в трехмерном пространстве. Трехмерную плоскость можно повернуть вокруг оси X, Y или Z. На рис. 9.13 показан поворот плоского изображения на 45° вокруг разных осей.

На рис. 9.13 элемент повернут вокруг своей центральной точки. С помощью свойств объекта PlaneProjection можно настроить параметры поворота.

- Свойство RotationX определяет угол поворота вокруг оси X. Свойство CenterOfRotationX содержит координату X центра поворота в относительных единицах. При значении 0 центр находится в крайней левой точке, 1 — крайней правой, 0.5 — посредине (это значение установлено по умолчанию).

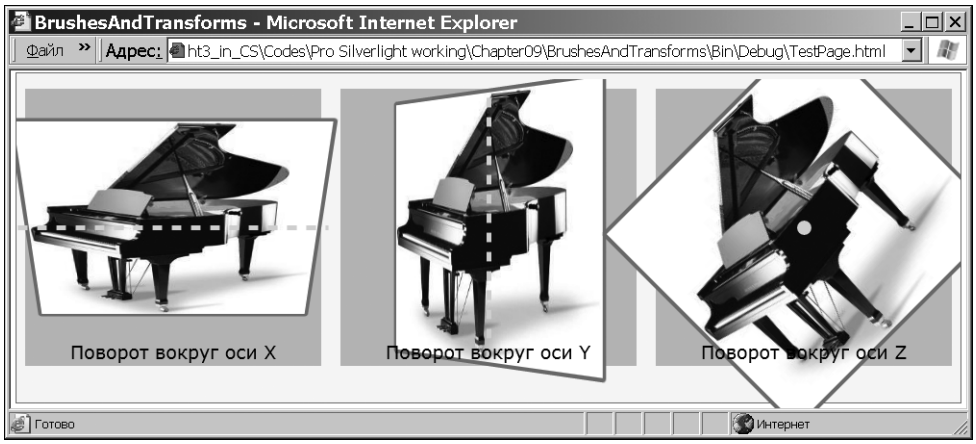

Рис. 9.13. Поворот изображения с помощью класса `PlaneProjection`

- Свойство `RotationY` определяет угол поворота вокруг оси Y. Свойство `CenterOfRotationY` содержит координату Y центра поворота в относительных единицах. При значении `0` центр находится в крайней верхней точке, `1` — крайней нижней, `0.5` — посредине (это значение установлено по умолчанию).
- Свойство `RotationZ` определяет угол поворота вокруг оси Z. Свойство `CenterOfRotationZ` содержит координату Z центра поворота в относительных единицах. При значении `0` центр находится посредине (это значение установлено по умолчанию), при положительном значении центр находится перед элементом, а при отрицательном — за элементом.

Во многих случаях свойства поворота — единственное, что вам нужно будет изменить при использовании объекта `PlaneProjection`. Однако элемент, кроме этого, можно еще и сместить. Существует два способа смещения.

- Свойства `GlobalOffsetX`, `GlobalOffsetY` и `GlobalOffsetZ` определяют перемещение элемента в экранных координатах **перед** операцией проецирования.
- Свойства `LocalOffsetX`, `LocalOffsetY` и `LocalOffsetZ` определяют перемещение элемента в преобразованных координатах **после** операции проецирования.

Когда элемент еще не повернут, глобальные и локальные свойства имеют одинаковые значения. Увеличение `GlobalOffsetX` или `LocalOffsetX` приведет к смещению элемента вправо. Предположим, объект был повернут вокруг оси Y с помощью свойства `RotationY` (рис. 9.14). Теперь увеличение `GlobalOffsetX` приведет к смещению отображенного содержимого вправо (как и в предыдущем случае, когда элемент не был повернут), однако увеличение `LocalOffsetX` приведет к смещению не точно вправо, а вдоль новой оси X, которая теперь указывает новое трехмерное направление. В результате, содержимое будет смещено вправо и вверх.

Применение проецирования

Операцию проецирования можно применить почти к любому элементу Silverlight, потому что каждый класс, производный от `UIElement`, имеет необходимое для этого свойство `Projection`. Чтобы получить для элемента эффект перспективы, нужно создать объект `PlaneProjection` и присвоить его свойству `Projection` в коде или разметке XAML.

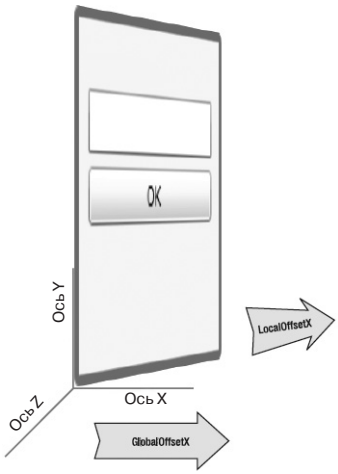

Рис. 9.14. Трансляция осей объектом `PlaneProjection`

Ниже приведена разметка, поворачивающая среднее изображение на рис. 9.13.

```
<Border BorderBrush="SlateGray" CornerRadius="2"
      BorderThickness="4">
  <Border.Projection>
    <PlaneProjection RotationY="45"></PlaneProjection>
  </Border.Projection>
  <Image Source="grandpiano.jpg"></Image>
</Border>
```

Как и обычное, перспективное преобразование выполняется после размещения элементов. На рис. 9.13 этот факт проиллюстрирован с помощью закрашенных областей, расположенных в исходной позиции. Элементы получили новые позиции, однако области с закрашенным фоном по-прежнему используются для вычисления позиций. Выполняется правило, применимое почти ко всем элементам: если элементы перекрываются, поверх других выводится элемент, объявленный последним (некоторые элементы, например `Canvas`, подчиняются не этому правилу, а значению свойства `ZIndex`).

Чтобы лучше понять, как взаимодействуют свойства объекта `PlaneProjection`, поэкспериментируйте с простым тестовым приложением (рис. 9.15). С помощью ползунков пользователь может поворачивать элемент вокруг осей X, Y и Z. Кроме того, элемент можно сместить локально или глобально вдоль оси X с помощью свойств `LocalOffsetX` и `GlobalOffsetX`, описанных выше.

Проецирование можно применить к любому элементу. Эта операция часто полезна при необходимости разместить в контейнере больше элементов. Приведенный выше пример интересен тем, что проецируемые элементы (кнопка и текстовое поле) и в повернутом состоянии продолжают взаимодействовать друг с другом стандартным образом: реагируют на щелчки мыши, получают фокус ввода и т.д. Ниже приведена разметка проецирования объекта `Border`.

```
<Border BorderBrush="SlateGray" CornerRadius="2"
      BorderThickness="4" Padding="10">
  <Border.Projection>
    <PlaneProjection x:Name="projection"></PlaneProjection>
  </Border.Projection>
  <StackPanel>
```

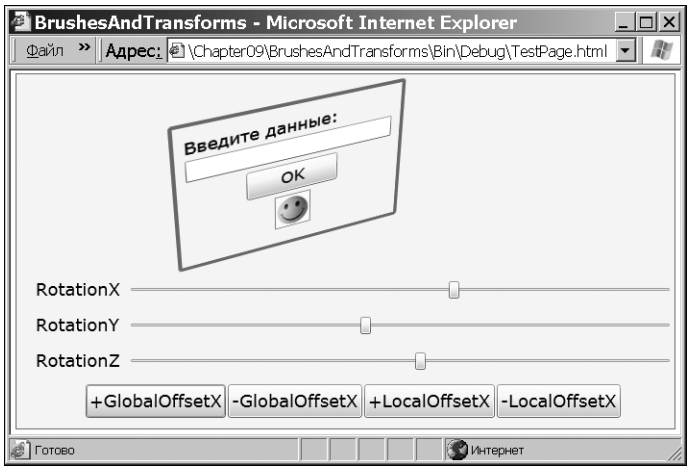

Рис. 9.15. Поворот и смещение элемента в трехмерном пространстве

```
    <TextBlock>Введите данные:</TextBlock>
    <TextBox></TextBox>
    <Button Margin="0,5" Content="OK"></Button>
    <Image Source="happyface.jpg" Stretch="None"></Image>
  </StackPanel>
</Border>
```

Объект `PlaneProjection` можно настроить с помощью кода C#, однако в данном примере используется связывание данных (см. главу 2). Объект `PlaneProjection` не является элементом, поэтому применить в нем выражение связывания нельзя. Выражение связывания вставлено в элемент `Slider`. Для передачи углов обратно в объект проецирования, когда пользователь перетаскивает ползунок, используется двустороннее связывание. Ниже приведена разметка ползунка X.

```
<TextBlock Margin="5">RotationX</TextBlock>
<Slider Grid.Column="1" Minimum="-180" Maximum="180"
 Value="{Binding RotationX, Mode=TwoWay,
       ElementName=projection}"></Slider>
```

Если повернуть элемент вокруг оси X или Y на угол более 90°, можно увидеть его с обратной стороны. Надстройка Silverlight считает, что элементы имеют прозрачную подложку, поэтому с обратной стороны символы выглядят, как зеркально отраженные (такое поведение существенно отличается от поведения трехмерных моделей WPF, в которых подложка по умолчанию считается пустой). В Silverlight интерактивные элементы, повернутые обратной стороной, продолжают работать, как обычно: реагируют на щелчки мышью, получают фокус, принимают текст с клавиатуры и т.д.

Раскрашивание пикселей

Одно из наиболее впечатляющих средств Silverlight 3 — *раскрашивание пикселей* (pixel shading) — используется для изменения внешнего вида элементов путем манипуляции пикселями перед выводом области содержимого на экран. Раскрашивание выполняется после операций преобразования и проецирования.

Раскрашивание в Silverlight — не менее мощное средство, чем графические надстройки Adobe Photoshop. Оно позволяет накладывать на элементы такие эффекты, как

размывание, сияние, водяная рябь, чеканка, обострение и т.д. Процедуру раскрашивания можно анимировать (см. главу 10) путем изменения параметров раскрашивания в реальном времени.

Каждый раскрашивающий объект (shader) представлен классом, производным от абстрактного класса `Effect`. Классы раскрашивания определены в пространстве имен `System.Windows.Media.Effects`. В надстройку Silverlight включены три базовых класса раскрашивания: `BlurEffect`, `DropShadowEffect` и `ShaderEffect`. В следующих разделах рассматривается каждый из них, а также способы интеграции в приложение дополнительных классов раскрашивания, предлагаемых независимыми поставщиками в различных библиотеках. Интеграция выполняется с помощью класса `ShaderEffect`.

Класс `BlurEffect`

Объект `BlurEffect` размывает визуальное содержимое элемента, как будто вы смотрите на него сквозь расфокусированную оптику. Интенсивность размывания определяется свойством `Radius`. По умолчанию оно равно `5`.

Для эффекта размывания нужно создать объект раскрашивания и присвоить его свойству `Effect` обрабатываемого элемента.

```
<Button Content="Размывание, Radius=2" Padding="5">
  <Button.Effect>
    <BlurEffect Radius="2"></BlurEffect>
  </Button.Effect>
</Button>
```

На рис. 9.16 показан эффект размывания, примененный к ряду стандартных кнопок при трех разных значениях `Radius`.

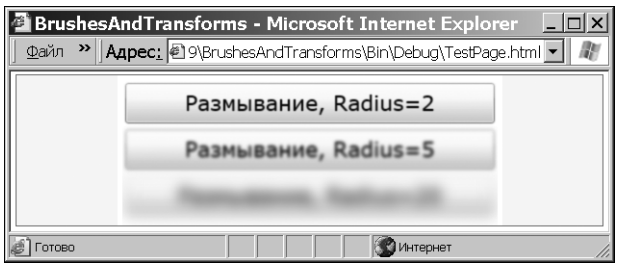

Рис. 9.16. Размытые кнопки

Класс `DropShadowEffect`

Объект `DropShadowEffect` создает эффект отбрасывания тени элементом. Для управления эффектом предназначены свойства, приведенные в табл. 9.4.

Таблица 9.4. Свойства класса `DropShadowEffect`

Имя	*Описание*
`Color` (Цвет)	Цвет тени; по умолчанию это свойство имеет значение `Black` (Черная)
`ShadowDepth` (Глубина тени)	Расстояние между тенью и содержимым в пикселях (по умолчанию равно `5`)

Окончание табл. 9.4

Имя	*Описание*
`BlurRadius` (Радиус размывания)	Степень размытости тени (как в объекте `BlurEffect`); по умолчанию равно `5`
`Opacity` (Непрозрачность)	Дробное число от `0` (тень полностью непрозрачная) до `1` (тень полностью прозрачная)
`Direction` (Направление)	Положение тени относительно содержимого, выраженное углом от 0 до 360˚; при значении 0 тень размещена справа; угол отсчитывается против часовой стрелки; по умолчанию свойство равно 315, при этом тень размещена справа снизу от элемента

На рис. 9.17 показано несколько эффектов отбрасывания тени, примененных к текстовым блокам. Ниже приведена разметка текстовых блоков с объектами `DropShadowEffect`.

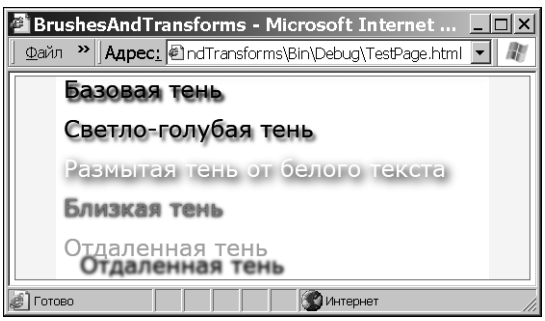

Рис. 9.17. Эффекты отбрасывания тени

```
<TextBlock FontSize="20" Margin="3">
  <TextBlock.Effect>
    <DropShadowEffect></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Базовая тень</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Margin="3">
  <TextBlock.Effect>
    <DropShadowEffect Color="SlateBlue"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Светло-голубая тень</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="White" Margin="3">
  <TextBlock.Effect>
    <DropShadowEffect BlurRadius="15"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Размытая тень от белого текста
  </TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="Magenta" Margin="3">
  <TextBlock.Effect>
```

```
    <DropShadowEffect ShadowDepth="0"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Близкая тень</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="LimeGreen" Margin="3">
  <TextBlock.Effect>
    <DropShadowEffect ShadowDepth="25"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Отдаленная тень</TextBlock.Text>
</TextBlock>
```

К сожалению, класса, группирующего эффекты, не существует. Поэтому за один раз к элементу можно применить только один эффект. Однако имитировать многие эффекты можно, применяя их к контейнеру более высокого уровня. Например, можно сначала применить тень к элементу `TextBlock`, а затем вложить его в панель `StackPanel` и применить к ней эффект размывания. В большинстве случаев использовать этот трюк не рекомендуется, потому что он существенно ухудшает производительность. Вместо этого поищите в библиотеках классов эффект, который сделает все, что вам нужно.

Класс `ShaderEffect`

Класс `ShaderEffect` не предоставляет готовых к использованию средств создания визуальных эффектов. Это абстрактный класс, предназначенный для наследования пользовательскими классами раскрашивания. С его помощью можно применить классы независимых поставщиков, позволяющие создавать эффекты, намного более мощные, чем размывание и отбрасывание тени.

Для многих покажется странным тот факт, что классы раскрашивания написаны не на C#, а на специализированном языке HLSL (High Level Shader Language — высокоуровневый язык раскрашивания), разработанном как часть технологии DirectX. Преимущества такого подхода очевидны: DirectX и HLSL используются уже много лет и разработчики графических приложений успели создать огромное количество процедур раскрашивания, которые можно использовать в Silverlight.

Для создания класса раскрашивания нужно разработать код HLSL. В первую очередь, установите пакет DirectX SDK (`www.msdn.microsoft.com/en-us/directx/aa937788.aspx`). Его достаточно для создания и компиляции кода HLSL в файл `.ps` (с помощью инструмента командной строки `fxc.exe`). Файл `.ps` необходим для пользовательского класса, наследующего `ShaderEffect`. Можете также применить бесплатный инструмент Shazzam (`www.shazzam-tool.com`), предоставляющий удобный редактор файлов HLSL. Кроме того, программа Shazzam содержит несколько образцов классов раскрашивания, которые можно использовать в качестве шаблонов.

Разработка файлов HLSL в данной книге не рассматривается. Рассмотрим лишь использование существующего файла HLSL. Когда код HLSL скомпилирован в файл `.ps`, его можно применить в проекте Silverlight. Добавьте файл `.ps` в существующий проект, выделите его в окне Solution Explorer (Проводник решений) и присвойте свойству Build Action (Операция построения) значение `Resource` (Ресурс). После этого нужно создать пользовательский класс, производный от `ShaderEffect` и ссылающийся на ресурс.

Предположим, процедура раскрашивания скомпилирована в файл `Effect.ps`. Ее можно использовать в следующем коде.

```
public class CustomEffect : ShaderEffect
{
    public CustomEffect()
    {
```

```
        // На ресурс ссылается URI,
        // синтаксис которого описан в главе 6
        // AssemblyName;component/ResourceFileName
        Uri pixelShaderUri = new
           Uri(«CustomEffectTest;component/Effect.ps",
          UriKind.Relative);

        // Загрузка файла .ps
        PixelShader = new PixelShader();
        PixelShader.UriSource = pixelShaderUri;
    }
}
```

После этого класс раскрашивания можно применить на любой странице. В первую очередь, сделайте пространство имен доступным, добавив его в код.

```
<UserControl
 xmlns:local="clr-namespace:CustomEffectTest" ...>
```

Создайте экземпляр пользовательского класса раскрашивания и присвойте его свойству `Effect` обрабатываемого элемента.

```
<Image>
  <Image.Effect>
    <local:CustomEffect></local:CustomEffect>
  </Image.Effect>
</Image>
```

Немного усложнив код, можно передать пользовательскому классу раскрашивания входные аргументы. Для этого нужно создать соответствующее зависимое свойство, вызвав статический метод `RegisterPicelShaderSamplerProperty()`.

Совет. Если вы не являетесь профессиональным графическим программистом, то вам лучше не писать код HLSL самому, а взять готовые образцы кода и немного модифицировать их, а еще лучше — приобрести компоненты независимых поставщиков, содержащие классы раскрашивания. Например, в Silverlight 3 прекрасно работает бесплатная библиотека WPF Pixel Shader Effects Library (`www.codeplex.com/wpffx`). Она содержит огромное количество визуальных эффектов, таких как водоворот, инверсия цветов, шум, кристаллизация, ржавчина, потеки, ореол, зернистость. Кроме того, вы можете анимировать раскрашивание (см. главу 10), в результате чего оно будет происходить в реальном времени на глазах у пользователя.

Класс `WriteableBitmap`

В главе 5 рассматривается вывод растровых изображений на экран с помощью элемента `Image`. Элемент `Image` всего лишь выводит готовое изображение, он не содержит средств создания или редактирования изображений.

Для этого предназначен класс `WriteableBitmap`, производный от класса `BitmapSource`. Объект `BitmapSource` используется для установки свойства `Image.Source` (непосредственно в коде C# или неявно в разметке XAML). Объект `BitmapSource` содержит доступную только для чтения рефлексию растровых данных, а объект `WriteableBitmap` — массив пикселей, доступных для изменения, что открывает ряд интересных возможностей.

Генерация растрового изображения

Наиболее прямолинейный способ использования класса `WriteableBitmap` состоит в создании всего растрового изображения вручную. На первый взгляд, этот процесс может показаться слишком трудоемким и скучным, однако его можно автоматизировать

с помощью кода C#. Данный способ полезен при создании фрагментов изображений и визуализации музыкальных файлов или научных данных. Необходимо создать процедуру, которая динамически рисует данные на основе коллекции двухмерных фигур (см. главу 8) или манипулирует непосредственно пикселями.

Для генерации растрового изображения нужно сначала создать в памяти *битовую карту* (bitmap) с помощью класса `WriteableBitmap` (иногда ее называют *растровой картой*). Ее ширина и высота на данном этапе задаются в пикселях. Ниже приведен код, создающий изображение с размерами текущей страницы.

```
WriteableBitmap wb = new WriteableBitmap(
  (int)this.ActualWidth,
  (int)this.ActualHeight);
```

Затем нужно заполнить пиксели. Для этого используется свойство `Pixels`, содержащее одномерный массив пикселей. Пиксели размещаются слева направо построчно. Строки заполняют изображение сверху вниз. Для получения конкретного пикселя используйте следующую формулу, в которой координата Y умножается на длину строки и к произведению добавляется координата X требуемого пикселя.

```
y*wb.PixelWidth+x
```

Например, для установки в коде пикселя с координатами (40, 100) можно применить следующий оператор.

```
wb.Pixels[100 * wb.PixelWidth + 40] = ...;
```

Цвет каждого пикселя задается с помощью одного целого числа без знака. Чтобы создать это число, нужно упаковать ряд величин (`alpha`, `red`, `green` и `blue`), каждая из которых может принимать значения от 0 до 255. Значение `alpha` определяет прозрачность пикселя, `red` — интенсивность красной компоненты, `green` — интенсивность зеленой компоненты и `blue` — интенсивность синей компоненты. Упаковать значения в одно число можно путем сдвига кодов чисел.

```
int alpha = 255;
int red = 100;
int green = 200;
int blue = 75;

int pixelColorValue =
  (alpha << 24) | (red << 16) | (green << 8) | (blue << 0);
wb.Pixels[pixelPosition] = pixelColorValue;
```

Ниже приведен код, проходящий по всем пикселям и заполняющий их случайными числами с примесью регулярной решетки (рис. 9.18).

```
Random rand = new Random();
for (int y = 0; y < wb.PixelHeight; y++)
{
   int red = 0; int green = 0; int blue = 0;

   // Изменение цвета для создания вертикальных линий
   // через каждые 5 пикселей и горизонтальных
   // линий через каждые 7 пикселей
   if ((x % 5 == 0) || (y % 7 == 0))
   {
      // Цвет выбирается случайным образом, но немного
      // зависит от координат X и Y, что создает
      // впечатление градиента
```

```
            red = (int)((double)y / wb.PixelHeight * 255);
            green = rand.Next(100, 255);
            blue = (int)((double)x / wb.PixelWidth * 255);
        }
        else
        {
            // Вычисление цвета для пикселей, не принадлежащих
            // линиям решетки
            red = (int)((double)x / wb.PixelWidth * 255);
            green = rand.Next(100, 255);
            blue = (int)((double)y / wb.PixelHeight * 255);
        }

        // Установка значений пикселей
        int pixelColorValue = (alpha << 24) | (red << 16) |
                              (green << 8) | (blue << 0);
        wb.Pixels[y * wb.PixelWidth + x] = pixelColorValue;
    }
```

Рис. 9.18. Динамически сгенерированное растровое изображение

По завершении процесса генерации нужно вывести изображение на экран. Легче всего это сделать с помощью элемента `Image`.

```
img.Source = wb;
```

После вывода сгенерированного изображения можно читать массив `Pixels` и модифицировать значения пикселей. Это дает возможность создать процедуры редактирования и тестирования растровой карты. Пример тестирования растровой карты в игре можно найти на сайте `www.tinyurl.com/mroklb`.

Копирование визуального содержимого

В предыдущем примере изображение генерировалось по пикселям с помощью кода. Класс `WriteableBitmap` предоставляет еще одну возможность создания изображения: скопировать визуальное содержимое существующего элемента. Другое название этой операции — *захват* (capturing). Сначала нужно создать экземпляр объекта `WriteableBitmap` обычным способом и определить его размеры, а затем скопировать

содержимое элемента в объект с помощью метода `Render()`. Метод `Render()` принимает два параметра: элемент, содержимое которого нужно захватить, и объект преобразования (или группа преобразований), с помощью которого при необходимости можно будет изменить изображение. Если преобразовывать захваченное содержимое не нужно, передайте значение `null`.

Ниже приведен код, который захватывает всю страницу Silverlight, имитируя копирование экрана. Скопировать весь экран нельзя, потому что доступ к содержимому за пределами области Silverlight создал бы проблемы безопасности.

```
// Поиск элемента самого высокого уровня
UserControl mainPage =
  (UserControl)Application.Current.RootVisual;

// Создание растровой карты
WriteableBitmap wb =
  new WriteableBitmap((int)mainPage.ActualWidth,
  (int)mainPage.ActualHeight);

// Копирование содержимого в растровую карту
wb.Render(mainPage, null);
wb.Invalidate();

// Вывод растрового изображения
img.Source = wb;
```

После вызова метода `Render()` нужно запустить метод `Invalidate()`, который прикажет растровой карте сгенерировать свое содержимое, задержав управление на время, необходимое для выполнения операции.

Заполнив растровую карту, ее можно вывести на экран с помощью объекта `Image`, как описано выше.

Совет. Метод `Writeablebitmap.Render()` особенно полезен с элементом `MediaElement`, потому что позволяет захватить кадр воспроизводимого видеофайла (см. главу 11).

Резюме

В этой главе рассмотрены расширенные средства двухмерной модели рисования Silverlight. При создании двухмерной графики важно понимать принципы использования графических инструментов, чтобы извлечь из них все, на что они способны. Например, изощренные эффекты можно создавать, изменяя кисти, используемые для заливки фигур, применяя объекты преобразований к графическим компонентам, изменяя прозрачность фигур и т.д. Вы можете добавлять, изменять и удалять любые индивидуальные компоненты графики. Рассмотренные способы рисования легко совмещаются со средствами анимации, обсуждаемыми в следующей главе. Например, средствами анимации можно плавно поворачивать фигуры с помощью свойства `Angle` объекта `RotateTransform`, постепенно прорисовывать элементы с помощью свойства `DrawingGroup.Opacity`, создавать анимированные градиенты с помощью кистей и классов раскрашивания. В следующей главе вы еще раз встретитесь с примерами использования рассмотренных технологий преобразования и раскрашивания.