Professional

Microsoft

Windows 程序设计（第6版 英文版）

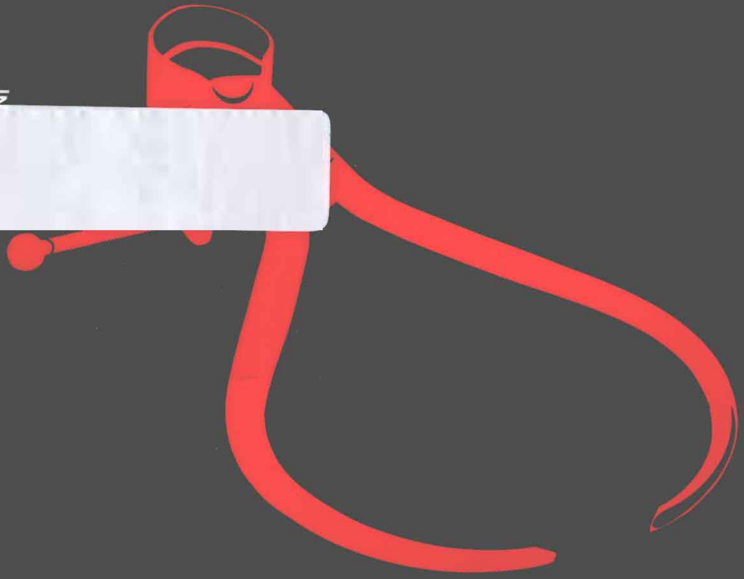
下册

[美] Charles Petzold 著

Programming Windows

Sixth Edition

- 使用C#和XAML
  编写Windows 8应用程序



人民邮电出版社
POSTS & TELECOM PRESS

Professional

Microsoft

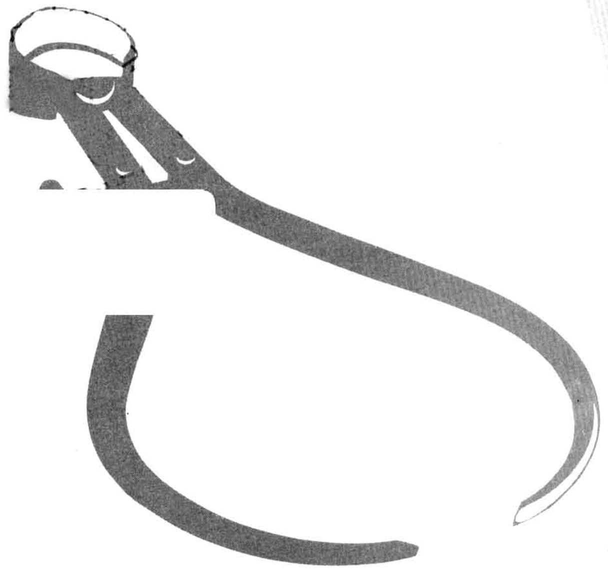
Windows 程序设计（第6版 英文版）

下册

[美] Charles Petzold 著

Programming Windows

Sixth Edition



人 民 邮 电 出 版 社

北 京

图书在版编目（ＣＩＰ）数据

Windows程序设计 ： 第6版 ： 英文 / （美） 佩措尔德
(Petzold,C.) 著. -- 北京 ： 人民邮电出版社, 2013.10
ISBN 978-7-115-32976-9

Ⅰ. ①W… Ⅱ. ①佩… Ⅲ. ①Windows操作系统—程序
设计—英文 Ⅳ. ①TP316.7

中国版本图书馆CIP数据核字(2013)第201789号

版 权 声 明

Charles Petzold, Programming Windows, Sixth Edition（ISBN: 9780735671768）.

Copyright 2013 by Posts & Telecom Press.

Original English language edition copyright ©2013 by Microsoft Corporation.

Published by arrangement with the original publisher, Microsoft Press, a division of Microsoft Corporation, Redmond, Washington, U.S.A. All rights reserved.

本书中文简体字版由美国微软出版社授权人民邮电出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

◆ 著　　　　[美] Charles Petzold
责任编辑　汪　振
责任印制　程彦红　焦志炜

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：71
字数：1684 千字　　　　2013 年10月第 1 版
印数：1- 2 500 册　　　　2013 年10月北京第 1 次印刷
著作权合同登记号　图字：01-2013-2652 号

定价：148.00 元（上、下册）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

Contents at a glance

此为试读，需要完整PDF请访问： www.ertongbook.com

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

PART II

Specialties

CHAPTER 13

Touch, Etc.

One of the most forward-looking aspects of the Windows Runtime is the consolidation of touch, mouse, and pen input. No longer is it necessary to add touch to an existing mouse-oriented application, or add some mouse support to a touch application. From the very beginning, the programmer treats all these forms of input in a fairly interchangeable manner. In accordance with the Windows Runtime programming interface, I will be using the word *pointer* to refer to input from touch, mouse, and the pen (also known as the stylus) when it's not necessary to distinguish the actual input device.

The best way to handle pointer input is through the existing Windows Runtime controls. As you've seen, standard controls such as *Button*, *Slider*, *ScrollViewer*, and *Thumb* all respond to pointer input and use that to deliver higher-level input to your application.

In some cases, however, the programmer needs to obtain actual pointer input, and for that purpose *UIElement* defines three different families of events:

- Eight low-level events beginning with the word *Pointer*
- Five higher-level events beginning with the word *Manipulation*
- *Tapped*, *RightTapped*, *DoubleTapped*, and *Holding* events

The *Control* class supplements these events with virtual protected methods beginning with the word *On* and followed by the event name.

To receive pointer input, a *FrameworkElement* derivative must have its *IsHitTestVisible* property set to *true* and its *Visibility* property set to *Visible*. A *Control* derivative must have its *IsEnabled* property set to *true*. The element must have some kind of graphical representation on the screen; a *Panel* derivative can have a *Transparent* background but not a *null* background.

All these events are associated with the element that is underneath your finger or mouse or pen at the time of the event. The only exception is when a pointer has been "captured" by an element, as you'll see later in this chapter.

If you need to track individual fingers, you'll want to use the *Pointer* events. Each event is accompanied by an ID number that uniquely identifies either an individual finger or pen touching the screen, or the mouse or pen. In this chapter I'll demonstrate how to use *Pointer* events for a finger-paint program and a piano keyboard (unfortunately without sound). Both these programs obviously need to handle simultaneous input from multiple fingers.

In a sense, the *Pointer* events are the only events you need. For example, if you wish to implement a feature that allows the user to stretch a photograph with two fingers, you can track *Pointer* events for those two fingers and measure how far they're moving apart. But calculations of this sort are provided for you in the *Manipulation* events. The *Manipulation* events consolidate multiple fingers into a single action, and they're ideal for moving, stretching, pinching, and rotating visual objects.

For some applications you might be puzzled whether to use *Pointer* or *Manipulation* events. The *Manipulation* events should probably be your first choice. Particularly if you think to yourself "I hope the user's not going to start using a second finger because I'll just have to ignore it," you probably want to use the *Manipulation* events. Then, if the user does use two or more fingers when only one finger is necessary, the multiple fingers will be averaged.

However, you'll also discover that the *Manipulation* events have an intrinsic lag. A finger touching the screen needs to move a bit before that finger is interpreted as contributing to a manipulation. *Manipulation* events are not fired if a finger taps or holds. Sometimes this lag will be enough to persuade you to use the *Pointer* events instead. The *XYSlider* custom control shown in this chapter is a case in point. The version shown in this chapter is written with *Manipulation* events because it wouldn't know what to do with extra fingers. But the lag time is a definite problem, so I have another version in Chapter 14, "Bitmaps," that uses *Pointer* events.

Pointer events are generated on a window level by the *CoreWindow* object, and you can derive *Manipulation* events on your own using the *GestureRecognizer*, but I'll be ignoring those facilities in this chapter and sticking with the events defined by *UIElement* and the virtual methods defined by *Control*. I also won't get into information about hardware input devices available from classes in the *Windows.Devices.Input* namespace.

Input from the pen has some special considerations involving the selection, erasing, and storage of pen strokes, as well as handwriting recognition. Those topics will be saved for Chapter 19, "Pen (Also Known as Stylus)." The Microsoft Surface tablet introduced in October 2012 does not support pen input.

A *Pointer* Roadmap

Of the eight *Pointer* events, five are very common. If you touch a finger to an enabled and visible *UIElement* derivative, move it, and lift it, these five *Pointer* events are generated in the following order:

- *PointerEntered*
- *PointerPressed*
- *PointerMoved* (multiple occurrences in the general case)
- *PointerReleased*
- *PointerExited*

此为试读，需要完整PDF请访问：www.ertongbook.com

A finger generates *Pointer* events only when the finger is touching the screen or when it has just been removed. There is no such thing as "hover" with touch.

The mouse is a little different. The mouse generates *PointerMoved* events even without the mouse button pressed. Suppose you move the mouse pointer to a particular element, press the button, move the mouse some more, release the button, and then move the mouse off the element. The element generates the following series of events:

- *PointerEntered*
- *PointerMoved* (multiple)
- *PointerPressed*
- *PointerMoved* (multiple)
- *PointerReleased*
- *PointerMoved* (multiple)
- *PointerExited*

Multiple *PointerPressed* and *PointerReleased* events can also be generated if the user presses and releases various mouse buttons.

Now let's try a pen. The element begins reacting to the pen before it actually touches the screen, so you'll first see a *PointerEntered* event followed by *PointerMoved*. As the pen touches the screen, a *PointerPressed* event is generated. Move the pen, and lift it. The element continues to fire *PointerMoved* events after *PointerReleased*, but it culminates with a *PointerExited* when the pen is moved farther away from the screen. It's the same sequence of events as the mouse.

When the user spins the mouse wheel, the following event is generated:

- *PointerWheelChanged*

The remaining two events are rarer:

- *PointerCaptureLost*
- *PointerCanceled*

I'll discuss pointer capture later in this chapter, at which time the *PointerCaptureLost* event becomes much more important.

I have never seen a *PointerCanceled* event even when I've unplugged the mouse from the computer, but the event exists to report an error of that sort.

All these events are accompanied by an instance of *PointerRoutedEventArgs*, defined in the *Windows.UI.Xaml.Input* namespace. (Watch out: There's also a *PointerEventArgs* class in the *Windows.UI.Core* namespace, but that's used for the processing of pointer input on the window level.) As the name of this class indicates, these *Pointer* events are all routed events that travel up the visual tree.

PointerRoutedEventArgs defines two properties common for routed events:

- *OriginalSource* indicates the element that raised the event.
- *Handled* lets you stop further routing of the event up the visual tree.

Lots of other information is available from the *PointerRoutedEventArgs* object. The following description covers only the highlights. The class also defines these members:

- *Pointer* property of type *Pointer*
- *KeyModifiers* property indicating the status of the Shift, Control, Menu (otherwise known as Alt), and Windows keys
- *GetCurrentPoint* method that returns a *PointerPoint* object

Watch out: Already we're dealing with classes named *Pointer* (defined in the *Windows.UI.Xaml.Input* namespace) and *PointerPoint* (defined in *Windows.UI.Input*).

The *Pointer* class has just four properties:

- *PointerId* property is an unsigned integer identifying the mouse, or an individual finger or pen.
- *PointerDeviceType* is an enumeration value *Touch*, *Mouse*, or *Pen*.
- *IsInRange* is a *bool* that indicates whether the device is in range of the screen.
- *IsInContact* is a *bool* indicating whether the finger or pen is touching the screen, or whether the mouse button is down.

The *PointerId* property is extremely important. This is what you use to track the movement of individual fingers. Almost always, a program that handles *Pointer* events will define a dictionary in which this *PointerId* property serves as a key.

The *GetCurrentPoint* method of *PointerRoutedEventArgs* sounds as if it returns the current coordinate location of the pointer, and it does, except that it also provides a whole lot more. Because it's convenient to get the location relative to a particular element, *GetCurrentPoint* accepts an argument of type *UIElement*. The *PointerPoint* object returned from this method duplicates some information from *Pointer* (the *PointerId* and *IsInContact* properties) and provides some other information:

- *Position* of type *Point*, the (*x*, *y*) location of the pointer at the time of the event
- *Timestamp* of type *ulong*
- *Properties* of type *PointerPointProperties* (defined in *Windows.UI.Input*)

The *Position* property is always relative to the upper-left corner of the element you pass to the *GetCurrentPoint* method.

PointerRoutedEventArgs also defines a method named *GetIntermediatePoints* that is similar to *GetCurrentPoint* except that it returns a collection of *PointerPoint* objects. Very often this collection has just one item—the same *PointerPoint* returned from *GetCurrentPoint*—but for the *PointerMoved* event there could be more than one, particularly if the event handler isn't very fast. I've particularly noticed *GetIntermediatePoints* returning multiple *PointerPoint* objects on the Microsoft Surface.

The *PointerPointProperties* class defines 22 properties that provide detailed information about the event, including which mouse buttons are pressed, whether the button on the pen barrel is pressed, how the pen is tilted, the contact rectangle of the finger with the screen (if that's available), the pressure of a finger or pen against the screen (if that's available), and *MouseWheelDelta*.

You can use as little or as much of this information as you need. Obviously, some of it will not be applicable to every pointer device and will therefore have default values.

A First Dab at Finger Painting

Perhaps the archetypal multitouch application is one that lets you paint with your fingers on the screen. You can write such a program handling just three *Pointer* events and examining just two properties from the event arguments, but I'm afraid the result has a flaw not quite compensated for by its simplicity.

The MainPage.xaml file of FingerPaint1 simply provides a name for the standard *Grid*:

Project: FingerPaint1 | **File:** MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Name="contentGrid"
          Background="{StaticResource ApplicationPageBackgroundThemeBrush}" />
</Page>
```

The very first thing that the code-behind file does is define a *Dictionary* with a key of type *uint*. I mentioned earlier that virtually every program that handles *Pointer* events has a *Dictionary* of this sort. The type of the items you store in the *Dictionary* is dependent on the application; sometimes an application will define a class or structure specifically for this purpose. In a rudimentary finger-painting application, each finger touching the screen will be drawing a unique *Polyline*, so the *Dictionary* can store that *Polyline* instance:

Project: FingerPaint1 | **File:** MainPage.xaml.cs (excerpt)

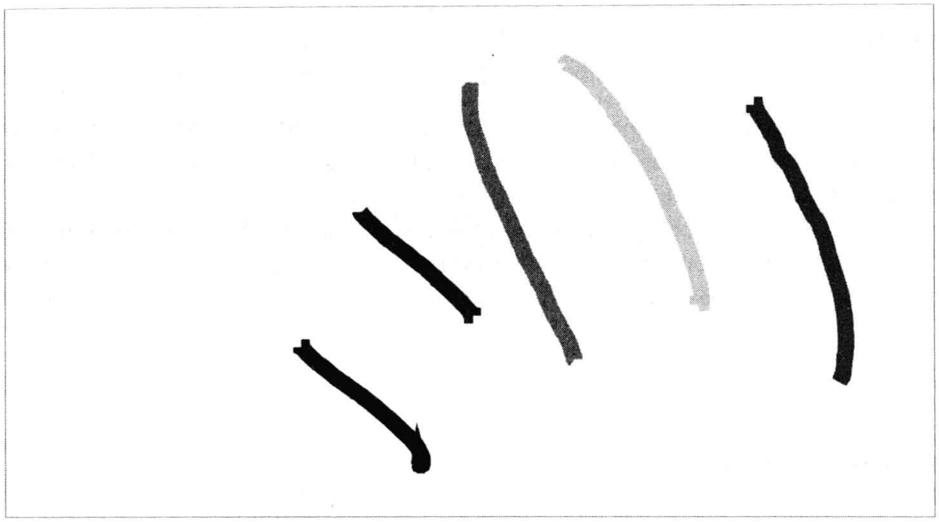
```
public sealed partial class MainPage : Page
{
    Dictionary<uint, Polyline> pointerDictionary = new Dictionary<uint, Polyline>();
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public MainPage()
    {
        this.InitializeComponent();
    }

    protected override void OnPointerPressed(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;
        Point point = args.GetCurrentPoint(this).Position;
        // Create random color
        rand.NextBytes(rgb);
        Color color = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);

        // Create Polyline
        Polyline polyline = new Polyline
        {
            Stroke = new SolidColorBrush(color),
            StrokeThickness = 24,
        };
        polyline.Points.Add(point);

        // Add to Grid
        contentGrid.Children.Add(polyline);

        // Add to dictionary
        pointerDictionary.Add(id, polyline);
        base.OnPointerPressed(args);
    }

    protected override void OnPointerMoved(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;
        Point point = args.GetCurrentPoint(this).Position;

        // If ID is in dictionary, add the point to the Polyline
        if (pointerDictionary.ContainsKey(id))
            pointerDictionary[id].Points.Add(point);

        base.OnPointerMoved(args);
    }

    protected override void OnPointerReleased(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;

        // If ID is in dictionary, remove it
        if (pointerDictionary.ContainsKey(id))
            pointerDictionary.Remove(id);

        base.OnPointerReleased(args);
    }
}
```

In the *OnPointerPressed* override, the program creates a *Polyline* and gives it a random color. The first point is the location of the pointer. The *Polyline* is added to the *Grid* and also to the dictionary.

When subsequent *OnPointerMoved* calls occur, the *PointerId* property identifies the finger, so the particular *Polyline* associated with that finger can be accessed from the dictionary and the new *Point* value can be added to the *Polyline*. Because it's the same instance as the *Polyline* in the *Grid*, the on-screen object will seem to grow in length as the finger moves.

The *OnPointerReleased* processing simply removes the entry from the dictionary. That particular *Polyline* is completed.

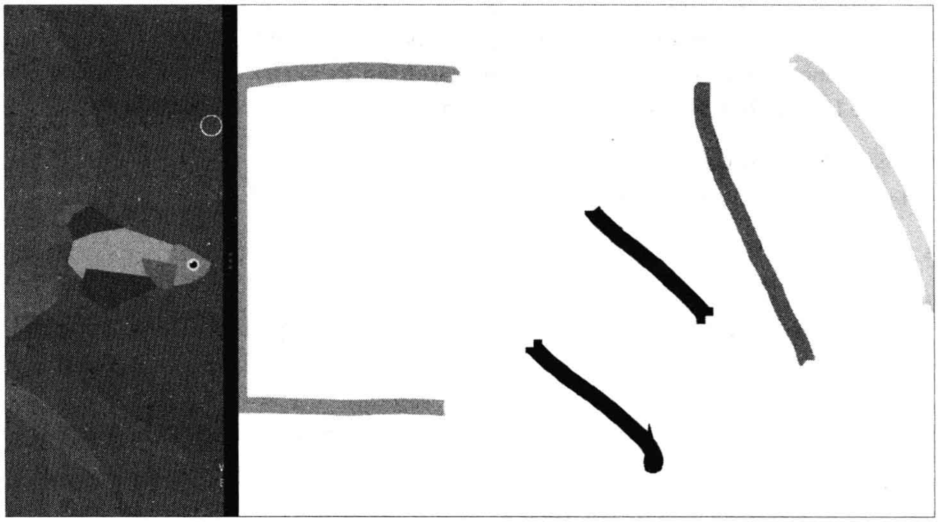
When you run the program, of course the first thing you'll want to do is sweep your whole hand across the screen like the glaciers that created the Finger Lakes in upstate New York.



Each finger paints its own polyline as a single series of connected points of a particular color, and you'll discover that you can use the mouse and pen as well.

I mentioned that this code has a flaw. The *OnPointerMoved* and *OnPointerReleased* overrides are very careful to check that the particular ID exists as a key in the dictionary before using it to access the dictionary. This is very important for mouse and pen processing because these devices generate *PointerMoved* events prior to *OnPointerPressed*.

But try this: Put the program in a snap mode, and with your finger, draw a line that goes outside the page and then back in.



Look at that straight line down the left side. That line is drawn when the finger reenters the page, and it indicates that the program doesn't get *PointerMoved* events during the time the finger strays outside. Try it with the mouse. Same thing.

Now try this: Using a finger, draw a line from the inside of the page to the outside and lift your finger. Now use your finger to draw inside the page again. This seems to work OK.

Now try it with the mouse. Press the mouse button over the FingerPaint1 page, move the mouse to outside the page, and release the mouse button. Now move the mouse to the FingerPaint1 page again. The program continues to draw the line even with the mouse button released! This is obviously wrong (but I'm sure you've seen programs that get "confused" like this). Now press the mouse button, and you'll generate an exception when the *OnPointerPressed* method attempts to add an entry to the dictionary using a key that already exists in the dictionary. Unlike touch or the pen, all mouse events have the same ID.

Let's fix these problems.

Capturing the Pointer

To allow me (and you) to get a better sense of the sequence of *Pointer* events, I wrote a program called PointerLog that logs all the *Pointer* events on the screen. The core of the program is a *UserControl* called *LoggerControl*. The *Grid* in the LoggerControl.xaml file has been given a name but is otherwise initially empty:

Project: PointerLog | **File:** LoggerControl.xaml (excerpt)

```
<UserControl ... >

    <Grid Name="contentGrid" Background="Transparent" />

</UserControl>
```

The code-behind file has overrides of all eight *Pointer* methods, all of which call a method named *Log* with the event name and event arguments. Like all *Pointer* programs, a *Dictionary* is defined, but the values in this one are not simple objects. Instead, I defined a nested class named *PointerInfo* right at the top of the *LoggerControl* class for storing per-finger information in this dictionary.

Project: PointerLog | **File:** LoggerControl.xaml.cs (excerpt)

```
public sealed partial class LoggerControl : UserControl
{
    class PointerInfo
    {
        public StackPanel stackPanel;
        public string repeatEvent;
        public TextBlock repeatTextBlock;
    };

    Dictionary<uint, PointerInfo> pointerDictionary = new Dictionary<uint, PointerInfo>();
```

此为试读，需要完整PDF请访问：www.ertongbook.com