* 1. 

Hands-On Lab

Multitouch WMTouch with MFC - Native

* 1. Lab version: 1.0.0
  2. Last updated: 12/23/2010

CONTENTS

[Overview 3](#_Toc243305605)

[Exercise 1: Build a Multitouch Application 5](#_Toc243305606)

[Task 1 – Prepare the Application 5](#_Toc243305607)

[Task 2 – Add Touch Support to the application 8](#_Toc243305608)

[Task 3 – Add the Stroke Source and Header Files to the Project, and Draw Lines with your Fingers 11](#_Toc243305609)

[Summary 16](#_Toc243305610)

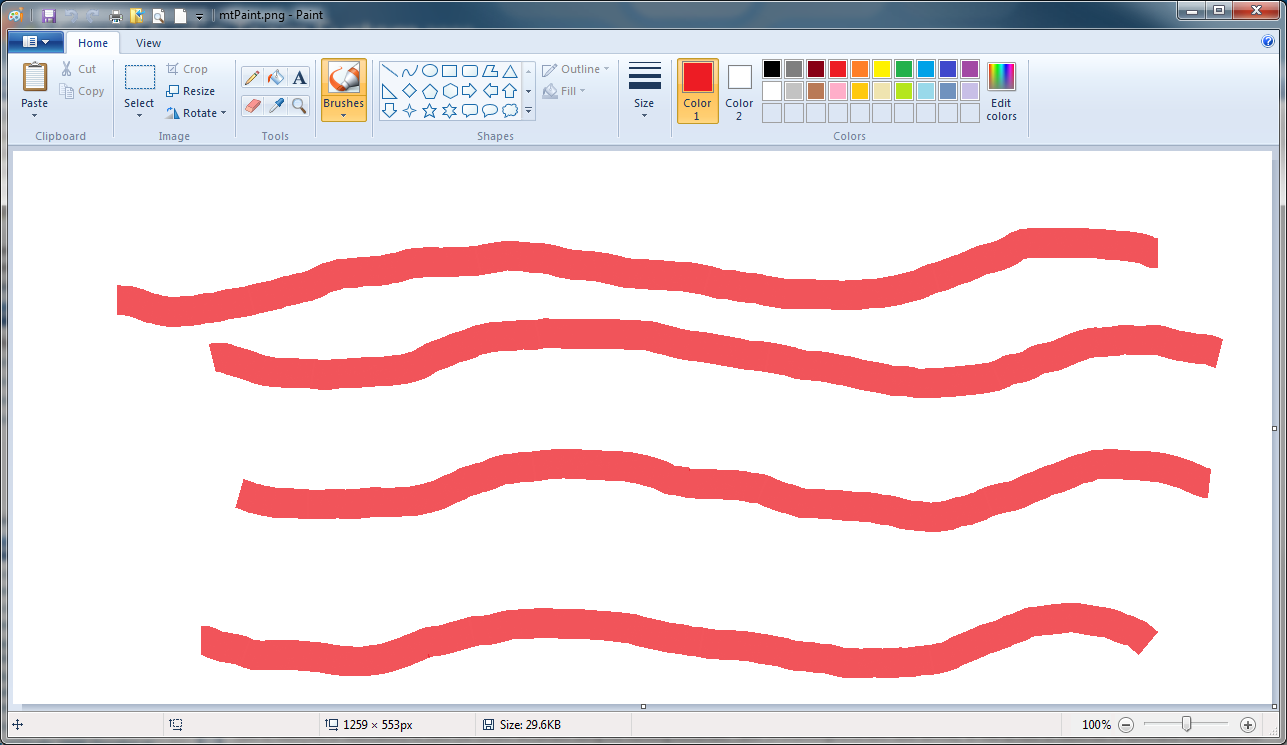
Overview

* 1. Windows 7 gives users the ability to manage applications with the touch of their fingers, using no intermediate device. This expands the stylus-based capabilities of tablet PCs. Unlike other pointing devices, this new capability allows multiple input events at the same time from different pointing locations, and it enables complex scenarios, such as managing applications with ten fingers or with multiple simultaneous users. However, to pull this off, we have to adapt our application's user interface and behavior to support this new input model.
  2. MFC in Visual Studio 2010 has added support for checking Multitouch hardware readiness and simplified the process of receiving touch events.

# Objectives

* 1. In this Hands-On Lab, you will learn how to manage Multitouch events, including:
  + Processing input from Windows Touch
  + Understanding the implications of manipulating multiple touch events simultaneously
  + Checking for Multitouch hardware existence and readiness

# System Requirements

* 1. You must have the following items to complete this lab:
  + Windows 7
  + Microsoft Visual Studio 2010
  + A Multitouch hardware device
  1. **Introduction**
  2. To create a Multitouch driven application you can choose one of three approaches: Good, Better, and Best.
  3. The “Good” approach is the easiest of the three. You should design your application user interface with touch ability in mind. Use large and clean Win32 based controls that make a natural interface for better user experience. Touch abilities such as scrolling come from the Win32 controls. There is no need for extra work. For example, try to scroll the document that you are reading now with your fingers! This is the “Good” approach.
  4. The "Better" approach lets the system receive various low-level touch events and passes the result of the heuristics that the system performs with these events to your application as “gestures”. For example, if the user makes a rotation movement on the screen, the system will issue a rotation gesture event with the rotation angle. Although the "Better" approach is easy to use, it has its limitations. Using gestures one cannot get Rotate, Translate, and Scale simultaneously. Also you cannot handle many different touch-based actions at the same time. For example two users that operate different areas of the Window.
  5. The “Best” approach is to read the low-level touch events as the input to the application. Applications like “Piano” or complex controls like multiple sliders that can be operated simultaneously are good examples. For example, run MS Paint, select a drawing tool from the gallery and draw with four of your fingers.
  6. 
  7. In this Hands-On Lab, you will mimic the new MS Paint Multitouch painting feature using the “Best” approach. We will read and use the raw touch events.

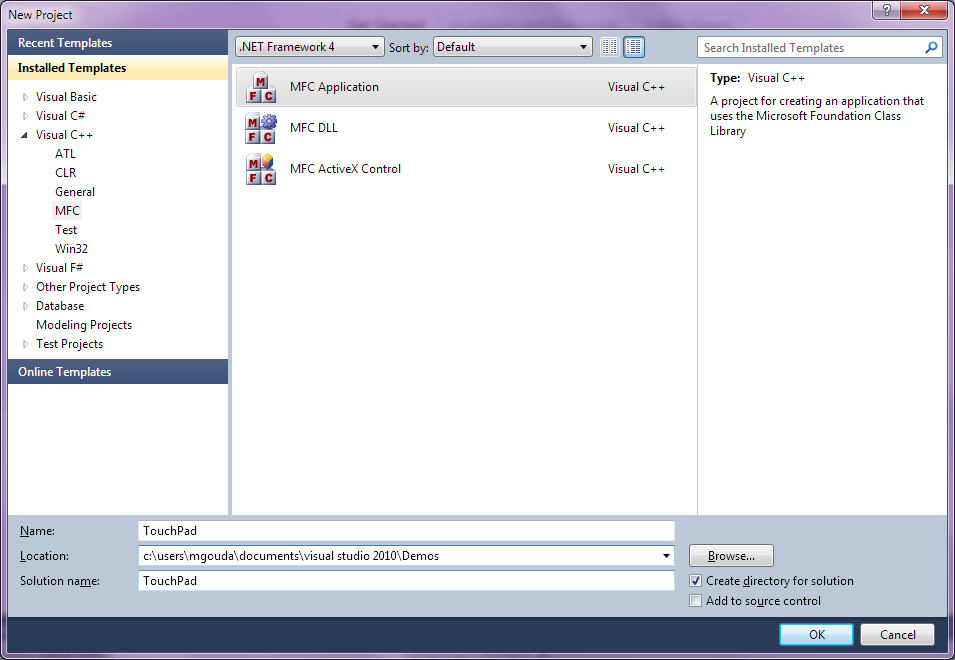
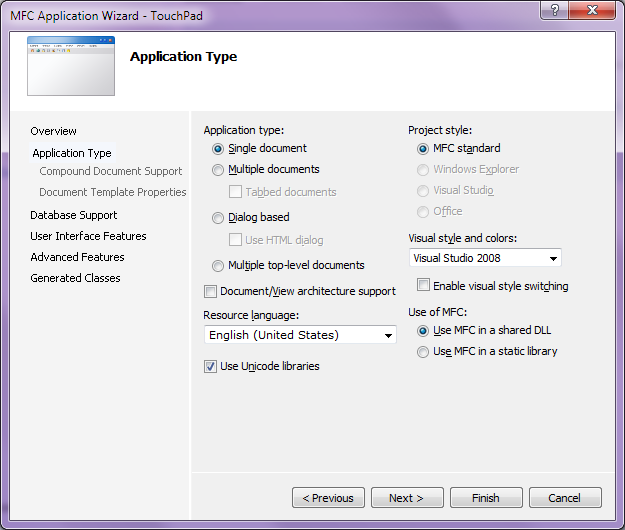
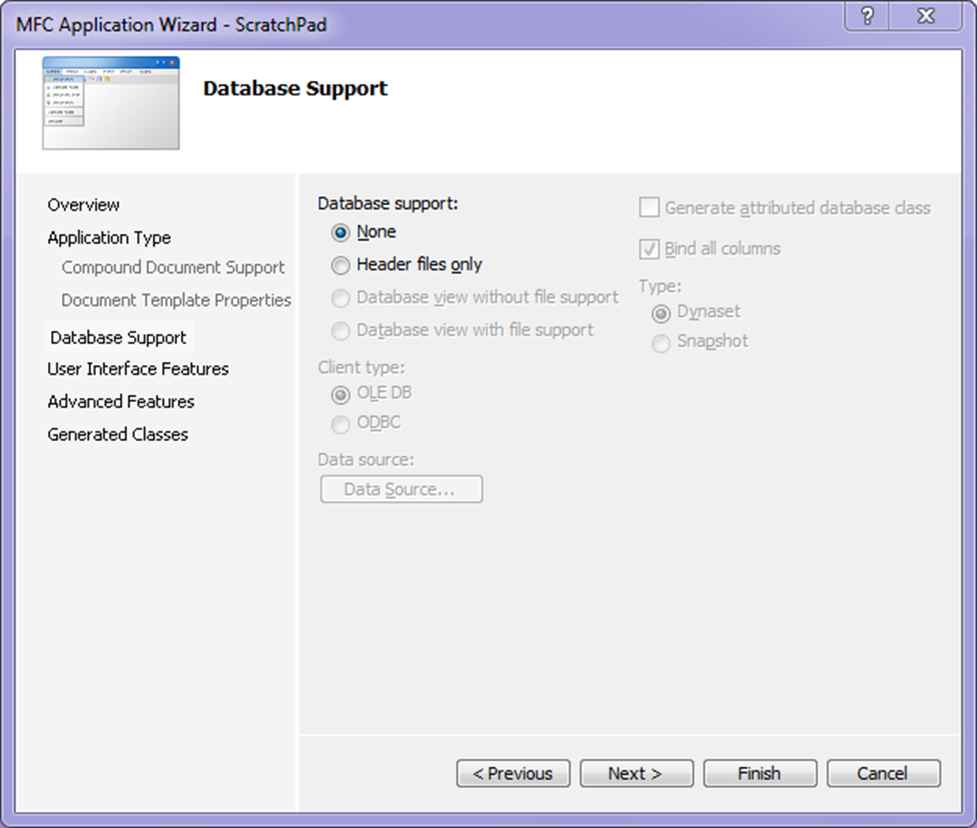
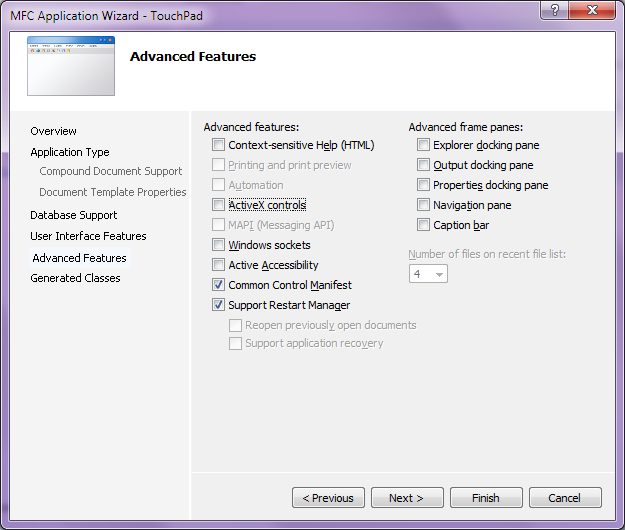
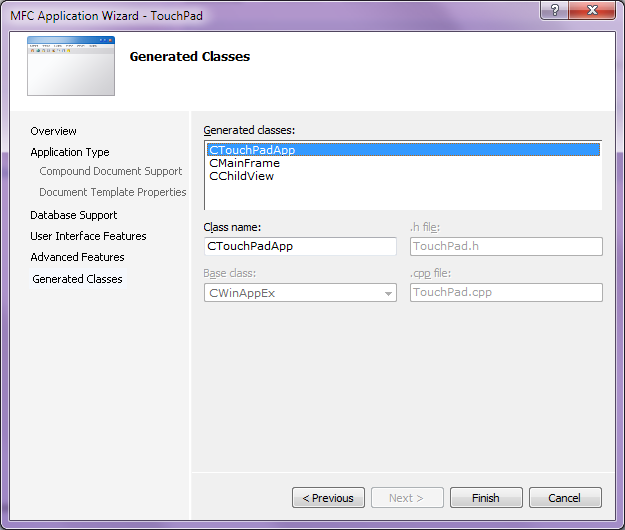
# About the Multitouch Scratchpad Application

The Multitouch Scratchpad application presents a simple window that allows simultaneously drawing of continuous lines with your fingers. While the folder **Source\*MFC\_WMTouchSource\Starter*** contains files needed for the exercise, ***Source\MFC\_WMTouchSource\Final*** contains the completed solution.

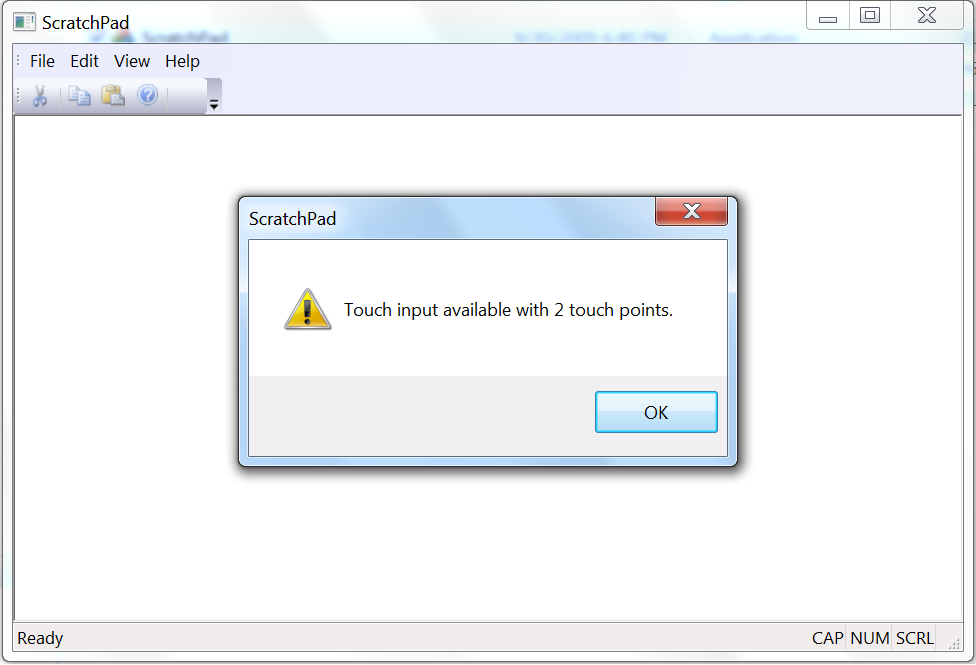


Exercise 1: Build a Multitouch Application

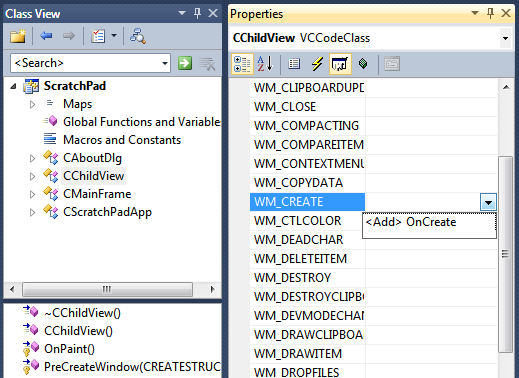
Task 1 – Prepare the Application

* 1. Start Visual Studio 2010
  2. Create a new MFC application project and give it the name ***ScratchPad***:
     1. 
  3. In the Application Type, select Single Document. To keep the application simple, unselect the other options in the dialog similar to the screens-shown below:
     1. 
     2. 
     3. 
  4. Continue clicking Next until you finally hit Finish:
     1. 

Task 2 – Add Touch Support to the application

* 1. The application that we are building requires touch-enabled hardware, so we need to make this check in the application.
  2. In Scratchpad.cpp, add the following check at the end of CScratchPadApp::InitInstance():
     1. C++
     2. BYTE digitizerStatus = (BYTE) GetSystemMetrics(SM\_DIGITIZER);
     3. if ((digitizerStatus & (0x80 + 0x40)) == 0) //Stack Ready + MultiTouch
     4. {
     5. AfxMessageBox(L"No touch input is currently available.");
     6. return FALSE;
     7. }
     8. BYTE nInputs = (BYTE) GetSystemMetrics(SM\_MAXIMUMTOUCHES);
     9. CString str;
     10. str.Format(L"Touch input available with %d touch points.", nInputs);
     11. AfxMessageBox(str);
     12. return TRUE;
  3. You can see that besides checking for touch availability and readiness we also find out the number of touch inputs that the hardware supports.
  4. Compile and run.
  5. Depending on the number of touch inputs you have on your machine, you should see output similar to this:
     1. 
  6. In order to register the application client view window to receive Touch messages, we need to call the MFC function CWnd::RegisterTouchWindow(). We’ll do so after the view has been created; i.e. in the OnCreate() event handler.

Switch to the Class View and Select the *CChildView* class.  
In the Properties page, go to the Message property sheet and navigate to WM\_CREATE, then add the OnCreate() message handler from the drop down box:

* + 1. 
  1. Inside the CChildView::OnCreate() handler append the following code to register touch input for the view window:
     1. C++
     2. if (!RegisterTouchWindow())
     3. {
     4. ASSERT(FALSE);
     5. }
     6. **Note:** Calling CWnd::RegisterTouchWindow() registers (and unregisters) a window as touch capable, allowing it to receive low-level WM\_TOUCH messages.
  2. Since we’ve registered the view to receive touch input, we must override the handler receiving each of the touch messages: CWnd::OnTouchInput().

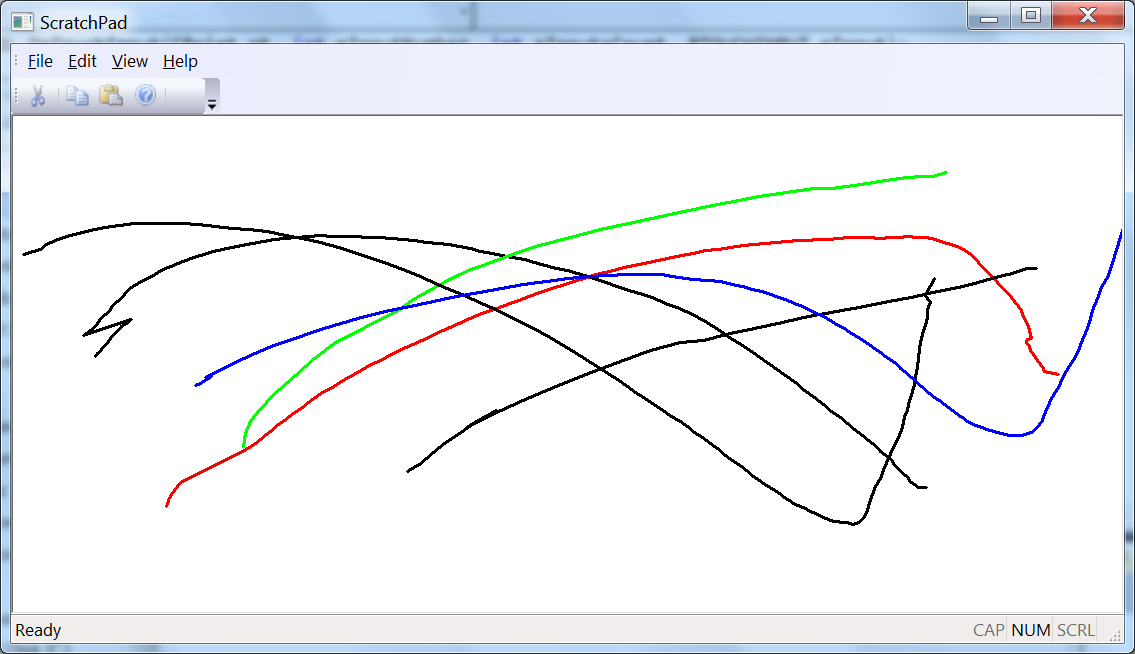
This handler receives a single input from Windows Touch and should return TRUE if the application processes this message; otherwise FALSE.

* 1. In *ChildView.h*, add this method declaration:
     1. C++
     2. // Overrides
     3. protected:
     4. virtual BOOL OnTouchInput(CPoint pt, int nInputNumber, int nInputsCount, PTOUCHINPUT pInput);
  2. And in ChildView.cpp, provide the corresponding implementation:
     1. C++
     2. BOOL CChildView::OnTouchInput(CPoint pt, int nInputNumber, int nInputsCount, PTOUCHINPUT pInput)
     3. {
     4. // TODO: Handle Tocuh input messages here
     5. return FALSE;
     6. }

Task 3 – Add the Stroke Source and Header Files to the Project, and Draw Lines with your Fingers

* 1. We would like to use our fingers as multiple input devices. We want to draw a line for each finger that touches the screen. To do that we are going to use two stroke collections. One collection holds the finished strokes (lines) and another collection holds the ongoing, currently painting lines. Each finger that touches the screen adds points to a stroke in the m\_StrkColDrawing collection. When we raise the finger from the screen, we move the finger's stroke from the m\_StrkColDrawing to the m\_StrkColFinished collection. In addition we want strokes to have different colors if a user is using two or more fingers on a multitouch monitor.
  2. In the Starter folder you will find two files: Stroke.h and Stroke.cpp. Copy them to the project folder and use “Add🡪 Existing item…” to add them to the project.
  3. Similarly, add StrokeCollection.h and StrokeCollection.cpp to the project.
  4. Include "Stroke.h" and "StrokeCollection.h" at the end of *StdAfx.h* header file.
     1. C++
     2. #include "Stroke.h"

#include "StrokeCollection.h"

* 1. Add these private member variable definitions to *ChildView.h*:
     1. C++
     2. private:
     3. int m\_iCurrColor; // The current stroke color
     4. CStrokeCollection m\_StrkColFinished; // The user finished entering strokes   
         // after user lifted his or her finger.
     5. CStrokeCollection m\_StrkColDrawing; // The Strokes collection the user is  
         // currently drawing.
  2. **Important:** we have to initialize the current color. We’ll do so in the CChildView constructor in ChildView.cpp:
     1. C++
     2. CChildView::CChildView() **: m\_iCurrColor(0)**
     3. {
     4. }
  3. To draw the finished collection, we add the following call to the end of the CChildView::OnPaint() handler. It will draw all finished strokes.
     1. C++
     2. m\_StrkColFinished.Draw(&dc);
  4. We need to process each Touch input message received, so we handle each of the events we’re interested in: *touch input down*, *move* and *up*.
  5. In *CChildView.h*, declare the following methods, which we’ll use to handle different touch input events:
     1. C++
     2. protected:
     3. // Handlers for different touch input events
     4. BOOL OnTouchInputDown(CPoint pt, PTOUCHINPUT pInput);
     5. BOOL OnTouchInputMove(CPoint pt, PTOUCHINPUT pInput);
     6. BOOL OnTouchInputUp(CPoint pt, PTOUCHINPUT pInput);
  6. In *CChildView.cpp*, add the implementation of each of the touch input handlers:
     1. C++
     2. BOOL CChildView::OnTouchInputDown(CPoint pt, PTOUCHINPUT pInput)
     3. {
     4. // Create new stroke and add point to it.
     5. COLORREF strokeColor = GetTouchColor((pInput->dwFlags & TOUCHEVENTF\_PRIMARY) != 0);
     6. CStroke\* pStrkNew = new CStroke(pInput->dwID, strokeColor);
     7. pStrkNew->Add(pt);
     8. // Add new stroke to the collection of strokes in drawing.
     9. m\_StrkColDrawing.Add(pStrkNew);
     11. return TRUE;
     12. }
     13. BOOL CChildView::OnTouchInputMove(CPoint pt, PTOUCHINPUT pInput)
     14. {
     15. // Find the stroke in the collection of the strokes in drawing.
     16. int strokeIndex = m\_StrkColDrawing.FindStrokeById(pInput->dwID);
     17. if (strokeIndex >= 0)
     18. {
     19. CStroke\* pStrk = m\_StrkColDrawing[strokeIndex];
     20. // Add contact point to the stroke
     21. pStrk->Add(pt);
     22. // Draw the last stroke
     23. pStrk->Draw(GetDC());
     24. }
     25. return TRUE;
     26. }
     27. BOOL CChildView::OnTouchInputUp(CPoint pt, PTOUCHINPUT pInput)
     28. {
     29. // Find the stroke in the collection of the strokes in drawing.
     30. int strokeIndex = m\_StrkColDrawing.FindStrokeById(pInput->dwID);
     31. if (strokeIndex >= 0)
     32. {
     33. CStroke\* pStrkCopy = m\_StrkColDrawing[strokeIndex];
     34. // Remove this stroke from the collection of strokes in drawing.
     35. m\_StrkColDrawing.RemoveAt(strokeIndex);
     36. // Add this stroke to the collection of finished strokes.
     37. m\_StrkColFinished.Add(pStrkCopy);
     38. }
     39. return TRUE;
     40. }
  7. In *CChildView.cpp*, modify the implementation of CChildView::OnTouchInput() to call each of the touch input handlers as needed:
     1. C++
     2. BOOL CChildView::OnTouchInput(CPoint pt, int nInputNumber, int nInputsCount, PTOUCHINPUT pInput)
     3. {
     4. if ((pInput->dwFlags & TOUCHEVENTF\_DOWN) == TOUCHEVENTF\_DOWN) // Touch Down event
     5. {
     6. return OnTouchInputDown(pt, pInput);
     7. }
     8. else if ((pInput->dwFlags & TOUCHEVENTF\_MOVE) == TOUCHEVENTF\_MOVE) // Touch Move event
     9. {
     10. return OnTouchInputMove(pt, pInput);
     11. }
     12. else if ((pInput->dwFlags & TOUCHEVENTF\_UP) == TOUCHEVENTF\_UP) // Touch Move event
     13. {
     14. return OnTouchInputUp(pt, pInput);
     15. }
     16. return FALSE;
     17. }
  8. Note that the method GetTouchColor() is called but is not yet implemented. This method is responsible for changing the color of the pen when a user moves more than one finger on the app window. Add this method’s declaration in CChildView.h:
     1. C++
     2. private:
     3. COLORREF GetTouchColor(bool bPrimaryContact);
  9. And here’s the implementation in CChildView.cpp:
     1. C++
     2. COLORREF CChildView::GetTouchColor(bool bPrimaryContact)
     3. {
     4. static COLORREF c\_arrColor[] = // Secondary colors array
     5. {
     6. RGB(255, 0, 0), // Red
     7. RGB(0, 255, 0), // Green
     8. RGB(0, 0, 255), // Blue
     9. RGB(0, 255, 255), // Cyan
     10. RGB(255, 0, 255), // Magenta
     11. RGB(255, 255, 0) // Yellow
     12. };
     13. COLORREF color;
     14. if (bPrimaryContact)
     15. {
     16. // The primary contact is drawn in black.
     17. color = RGB(0,0,0); // Black
     18. }
     19. else
     20. {
     21. // Take current secondary color.
     22. color = c\_arrColor[m\_iCurrColor];
     23. // Move to the next color in the array.
     24. m\_iCurrColor = (m\_iCurrColor + 1) % (sizeof(c\_arrColor)/sizeof(c\_arrColor[0]));
     25. }
     26. return color;
     27. }
  10. Finally, since we’ve dynamically created a number of strokes, we need to make sure each of those is destroyed before the application exits, so we include the following in the CChildView‘s destructor:
      1. C++
      2. CChildView::~CChildView()
      3. {
      4. for (int i = 0; i < m\_StrkColDrawing.GetCount(); ++i)
      5. {
      6. delete m\_StrkColDrawing[i];
      7. }
      8. for (int i = 0; i < m\_StrkColFinished.GetCount(); ++i)
      9. {
      10. delete m\_StrkColFinished[i];
      11. }
      12. }
  11. You’re now done with coding, so you can start experimenting with the application you’ve just implemented.
  12. Build and run the application. It should look like this:
      1. 

Summary

* 1. In this lab, you have learned how to handle touch input in an MFC application. You’ve seen how to test for the existence of Multitouch hardware, and how to configure a window to receive touch input. Also, you have seen how to extract input from a message and how the system correlates touch id with touch input.
  2. Enjoy and keep in touch!