Hands-On Lab

Touch Input and Hybrid Apps

Lab version: 1.0.0

Last updated: 6/20/2012



Contents

[Overview 3](#_Toc327918230)

[Exercise 1: Simple Gestures 4](#_Toc327918231)

[Task 1 – Build the Starter Application 4](#_Toc327918232)

[Task 2 – Handle MSGestureTap Events from the Opening Screen 5](#_Toc327918233)

[Task 3 – Handle MSGestureTap Events from the Image 7](#_Toc327918234)

[Exercise 2: Panning and Zooming 8](#_Toc327918235)

[Task 1 – Add Support for Panning 8](#_Toc327918236)

[Task 2 – Handle MSGestureDoubleTap Events 10](#_Toc327918237)

[Task 3 – Add Support for Pinch-Zooming 10](#_Toc327918238)

[Task 4 – Add Mousewheel Support 12](#_Toc327918239)

[Exercise 3: Hybrid Apps 13](#_Toc327918240)

[Task 1 – Write a WinRT Component in C++ 14](#_Toc327918241)

[Task 2 – Call the WinRT Component from JavaScript 15](#_Toc327918242)

[Exercise 4: Pointer Input 17](#_Toc327918243)

[Task 1 - Handle Pointer Events 18](#_Toc327918244)

[Task 2 – Modify the Application Bar 21](#_Toc327918245)

[Task 3 – Test the Results 22](#_Toc327918246)

[Summary 22](#_Toc327918247)

Overview

* 1. Windows 8 supports input from a variety of human-interface devices, including mice and keyboards, but it makes touch input a first-class citizen. Touch APIs built into the operating system enable developers to build rich, intuitive, and responsive touch interfaces into Metro-style apps. This is particularly important on tablets, where the user is likely to have no mouse attached.
  2. In this lab, you’ll take a preexisting photo-editing app named Contoso Photo and add touch support to turn it into a fully functional application. You’ll add support for simple gestures such as taps and double-taps, and you’ll build in support for pinch-zooms as well as for dragging and panning. In addition, you’ll write a WinRT component in C++ and call it from JavaScript to help perform image processing chores. Sound appealing? Then let’s dive into the world of touch and see what it takes to build a great touch-enabled Metro-style app.

# Objectives

* 1. This lab will show you how to:
  + Respond to simple gestures such as taps
  + Implement dragging and panning with a single finger
  + Implement pinch-zooms (and unzooms) using two fingers
  + Implement mousewheel zoom for non-touch devices
  + Call WinRT components written in C++ from JavaScript

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Windows 8 Release Preview
  + Microsoft Visual Studio 2012 RC

# Setup

* 1. You must perform the following steps to prepare your computer for this lab:
  2. Install the Microsoft Windows 8 Release Preview
  3. Install the Microsoft Visual Studio 2012 RC

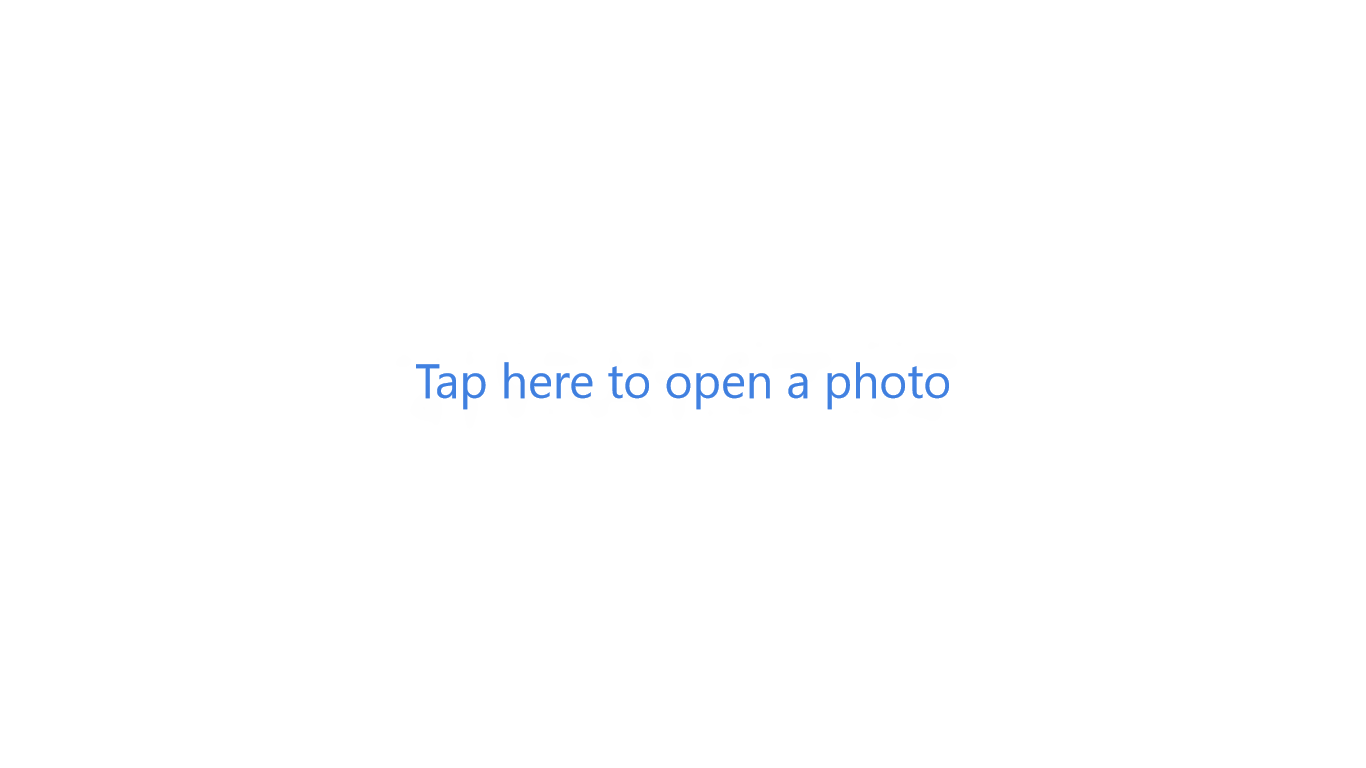
# Exercises

* 1. This Hands-On Lab comprises the following exercises:
  2. Simple Gestures
  3. Panning and Zooming
  4. Hybrid Apps
  5. Pointer Input
  6. Estimated time to complete this lab: **30 to 40 minutes**.

Exercise 1: Simple Gestures

1. In this exercise, you’ll start with an existing application – Contoso Photo – that has a lot of features in place already but that is missing touch support and add support for tap gestures.

Task 1 – Build the Starter Application

* 1. Let’s begin by retrieving the starter application from the starting materials and taking it for a test drive.
  2. Copy the solution named ContosoPhoto from the starting materials to your local hard drive.
  3. Open ContosoPhoto in Visual Studio.
  4. Press F5 to launch the application in the debugger. You should see the opening screen in Figure 1.
     1. 
     2. Figure 1
     3. Contoso Photo’s opening screen
  5. Tap the opening message: “Tap here to open a photo.” Does anything happen?
  6. Return to Visual Studio and stop debugging.

Task 2 – Handle MSGestureTap Events from the Opening Screen

* 1. The message “Tap here to open a photo” is contained in a DIV declared in default.html. Let’s add code to recognize when the DIV is tapped so we can clear it out and get to the main screen.
     1. **Note:** JavaScript developers are familiar with DOM events such as click, mousedown, and mouseup, which signify actions performed with the mouse. Windows 8 fires additional events from the DOM to make pointer input – that is, input performed with a mouse, stylus, or finger – as simple as possible. For example, when a finger touches the screen, an MSPointerDown event fires. When the finger moves across the screen, MSPointerMove events fire. And when the finger lifts off the screen, an MSPointerUp event fires. The same events fire if a stylus or mouse is used to perform the input.
     2. But it gets even better. To make processing gestures easy, Windows 8 provides the MSGesture class. If you forward MSPointerDown events from DOM elements to an instance of MSGesture, those DOM elements then fire events such as MSGestureTap, MSGestureDoubleTap, and MSGestureChange. Recognizing when a DOM element such as a DIV or an image is tapped is as simple as registering a handler for MSGestureTap events fired by that element.
     3. Taps are simple gestures executed with a single finger, but MSGestureTap events fire if an element is tapped with the mouse or a stylus as well. Consequently, when you handle these events, you get mouse and stylus support for free. Moreover, MSGestureTap events aren’t equivalent to mousedown events. An MSGesture Tap event only fires if the pointing device makes and breaks contact with an element within about half a second.
  2. Open default.js in the project’s js folder and add the following statement to the other vars at the top of the file:
     1. JavaScript
     2. var \_gesture = new MSGesture();
  3. Now add the following statements to the end of the app.onactivated handler, after the call to WinJS.UI.processAll:
     1. JavaScript
     2. // Register handlers for MSPointerDown events for gesture support
     3. document.querySelector("#message").addEventListener("MSPointerDown", onAssignPointer);
     4. document.querySelector("#main").addEventListener("MSPointerDown", onAssignPointer);
     5. \_mainCanvas.addEventListener("MSPointerDown", onAssignPointer);
     6. // Register handlers for MSGesture events for gesture recognition
     7. document.querySelector("#message").addEventListener("MSGestureTap", onOpeningMessageTapped);
  4. Add the following event handlers to default.js:
     1. JavaScript
     2. function onAssignPointer(e) {
     3. try {
     4. \_gesture.target = e.target;
     5. } catch (err) { }
     6. if (e.target == \_gesture.target) {
     7. \_gesture.addPointer(e.pointerId);
     8. }
     9. }
     10. function onOpeningMessageTapped(e) {
     11. onOpenButtonClicked(e);
     12. }
  5. Press F5 to launch the application.
  6. Tap “Tap here to open a photo.”
  7. In the ensuing FileOpenPicker, select an image file and click the Open button. (The starting materials contain an image file named Amy.jpg that is perfect for the exercises in this lab.)
  8. Confirm that the image you selected appears on the screen, as shown in Figure 2.
     1. 
     2. Figure 2
     3. Contoso Photo after loading an image
  9. Return to Visual Studio and stop debugging.

Task 3 – Handle MSGestureTap Events from the Image

* 1. Contoso Photo already contains logic to remove red eyes from photos; that logic is contained in a function named removeRedEye in default.js. Let’s allow users to fix red eyes in a photo simply by tapping them with a finger, a stylus, or a mouse.
  2. In default.js, add the following statement to the end of the app.onactivated handler:
     1. JavaScript
     2. \_mainCanvas.addEventListener("MSGestureTap", onImageTapped);
  3. Now add the following event handler to default.js:
     1. JavaScript
     2. function onImageTapped(e) {
     3. if (\_bitmap !== null) {
     4. var x, y;
     5. // Get the click coordinates
     6. x = e.offsetX;
     7. y = e.offsetY;
     8. // Translate click coordinates into image coordinates
     9. x = Math.round(x \* \_hiddenCanvas.width / \_mainCanvas.width);
     10. y = Math.round(y \* \_hiddenCanvas.height / \_mainCanvas.height);
     11. // Fix red eye in region under cursor
     12. removeRedEye(x, y);
     13. // Enable the Save button
     14. \_saveButton.removeAttribute("disabled");
     15. }
     16. }
  4. Press F5 to launch the application.
  5. Load a photo that contains red eyes, such as the Amy.jpg file supplied with the starting materials.
  6. Tap each of the red eyes and verify that the red goes away.
     1. **Note: I**f the red eyes don’t disappear, it’s probably because the eyes are small and your finger is comparatively large. If all else fails, try tapping around the eye a few times, or use a mouse or stylus to click precisely on the red part of the eye.
     2. Also, although it’s not part of the lab exercises, Contoso Photo has an application bar with a Save button that you can click to save an edited photo. Try it!
  7. Return to Visual Studio and stop debugging.

Exercise 2: Panning and Zooming

1. We are off to a good start, but Contoso Photo would be immeasurably more useful if users could zoom in and out on photos as they edit them. In this exercise, you will add support for pinch gestures, allowing users to zoom in and out as desired. You will also add dragging and panning support so users can move around within a zoomed photo.

Task 1 – Add Support for Panning

* 1. We will start by using the MSGestureChange events that fire as a pointing device moves around over a DOM element to allow the user to drag the image and reposition it on the screen. This will be only marginally useful at the moment, but it will be critically important once we add zoom support in the next task.
  2. Add the following statements to the other vars at the top of default.js:
     1. JavaScript
     2. var \_dx = 0;

var \_dy = 0;

* 1. Add the following statements to the end of the app.onactivated handler:
     1. JavaScript
     2. \_mainCanvas.addEventListener("MSGestureChange", onGestureChange);

\_mainCanvas.addEventListener("MSGestureEnd", onGestureEnd);

**Note:** In a Metro style app, MSGestureChange events fire when a DOM element is manipulated with one or two fingers. MSGestureChange event handlers receive information that can be used to move the DOM element if one finger is in contact with the screen, or scale it if two fingers are in contact and are moving together or apart. The related MSGestureStart and MSGestureEnd events signify that a manipulation has started or ended.

* 1. Add the following event handler to default.js:
     1. JavaScript
     2. function onGestureChange(e) {
     3. \_dx += e.translationX;
     4. \_dy += e.translationY;
     5. e.target.style.msTransform = "translate(" + \_dx + "px," + \_dy + "px)";
     6. }
     7. **Note:** The statement that sets the msTransform property moves the image by applying a CSS transform to it. You can read all about CSS transforms at http://www.w3.org/TR/css3-2d-transforms/.
  2. Find the image.onload handler inside the onOpenButtonClicked function. Add the three lines after the comment that reads “Reset the transform and related variables” – to reset the transform when a new image is loaded:
     1. JavaScript
     2. image.onload = function () {
     3. // Set the width and height of the canvas based on the image size
     4. var width = image.width;
     5. var height = image.height;
     6. var ratio = height / width;
     7. \_mainCanvas.width = Math.min(800, width);
     8. \_mainCanvas.height = \_mainCanvas.width \* ratio;
     9. // Reset the transform and related variables
     10. \_dx = 0;
     11. \_dy = 0;
     12. \_mainCanvas.style.msTransform = "translate(0px,0px)";
     13. // Convert the image into an ImageData object by drawing it to
     14. // the hidden canvas and calling getImageData on that canvas
     15. \_hiddenCanvas.width = image.width;
     16. \_hiddenCanvas.height = image.height;
     17. \_hiddenContext.drawImage(image, 0, 0, \_hiddenCanvas.width, \_hiddenCanvas.height);
     18. \_bitmap = \_hiddenContext.getImageData(0, 0, \_hiddenCanvas.width, \_hiddenCanvas.height);
     19. // Transfer (and scale) the image from the hidden canvas to the main canvas
     20. \_mainContext.drawImage(\_hiddenCanvas, 0, 0, \_mainCanvas.width, \_mainCanvas.height);
     21. // Disable the Save button
     22. \_saveButton.setAttribute("disabled", "");
     23. }
  3. Press F5 to launch the application.
  4. Load an image file.
  5. Use a finger (or mouse or stylus if you prefer) to drag the photo around and verify that the photo follows your finger.
  6. Now flick the photo toward the edge of the screen. Observe that it continues moving even after your finger breaks contact. That effect is called *inertia*, and it comes for free with MSGesture events.
  7. Return to Visual Studio and stop debugging.

Task 2 – Handle MSGestureDoubleTap Events

* 1. The downside to built-in inertia is that if you flick the photo hard enough, it flies off the screen! Before we go further, let’s use MSGestureDoubleTap events to recenter the photo on the screen.
  2. In default.js, add the following statement to the end of app.onactivated:
     1. JavaScript
     2. document.querySelector("#main").addEventListener("MSGestureDoubleTap", onDoubleTap);
     3. **Note:** You’re processing this event on the DIV that contains the image so you can catch a double-tap anywhere on the screen, even if it’s not over the photo.
  3. Add the following event handler to default.js:
     1. JavaScript
     2. function onDoubleTap(e) {
     3. \_dx = 0;
     4. \_dy = 0;
     5. \_mainCanvas.style.msTransform = "translate(0px,0px)";
     6. }
  4. Launch the application, load a photo, and flick it off the screen.
  5. Double-tap the screen and verify that the photo returns to the center.
  6. Return to Visual Studio and stop debugging.

Task 3 – Add Support for Pinch-Zooming

* 1. It is difficult to fix red eyes with any precision right now without resorting to a mouse or stylus because fingers are imprecise pointing devices. Let’s fix that by allowing the user to zoom in and out using pinch gestures. It’s easy to do in Windows 8, thanks to the events that you’ve already written handlers for.
  2. In default.js, add the following statement after the statements that declare variables named \_dx and \_dy near the top of the file:
     1. JavaScript
     2. var \_scale = 1;
  3. Find the onGestureChange function you added in Task 1 and modify it as shown below:
     1. JavaScript
     2. function onGestureChange(e) {
     3. \_dx += e.translationX;
     4. \_dy += e.translationY;
     5. \_scale = \_scale \* e.scale;
     6. \_scale = Math.min(\_scale, 4);
     7. \_scale = Math.max(\_scale, 1);
     8. e.target.style.msTransform = "scale(" + \_scale + ") translate(" + \_dx + "px," + \_dy + "px)";
     9. }
     10. **Note:** MSGestureChange events fire as a finger moves across a DOM element, and when two fingers move over a DOM element. When only one finger is in contact with the screen, e.translationX and e.translationY contain nonzero values specifying how much the finger has moved since the last MSGestureChange event, while e.scale contains the value 1.0. When two fingers in contact with the screen move relative to one another, e.scale holds a distance ratio indicating how much the fingers have moved together or apart since the last MSGestureChange event.
  4. Find the onDoubleTap function you added in Task 2 and modify it as follows:
     1. JavaScript
     2. function onDoubleTap(e) {
     3. // Reset the transform and related variables
     4. \_dx = 0;
     5. \_dy = 0;
     6. \_scale = 1;
     7. \_mainCanvas.style.msTransform = "scale(1) translate(0px,0px)";
     8. }
  5. Finally, modify the two statements below the statement “\_dy = 0” in the image.onload handler to reset the transform when the user opens a new image:
     1. JavaScript
     2. image.onload = function () {
     3. // Set the width and height of the canvas based on the image size
     4. var width = image.width;
     5. var height = image.height;
     6. var ratio = height / width;
     7. \_mainCanvas.width = Math.min(800, width);
     8. \_mainCanvas.height = \_mainCanvas.width \* ratio;
     9. // Reset the transform and related variables
     10. \_dx = 0;
     11. \_dy = 0;
     12. \_scale = 1;
     13. \_mainCanvas.style.msTransform = "scale(1) translate(0px,0px)";
     14. // Convert the image into an ImageData object by drawing it to
     15. // the hidden canvas and calling getImageData on that canvas
     16. \_hiddenCanvas.width = image.width;
     17. \_hiddenCanvas.height = image.height;
     18. \_hiddenContext.drawImage(image, 0, 0, \_hiddenCanvas.width, \_hiddenCanvas.height);
     19. \_bitmap = \_hiddenContext.getImageData(0, 0, \_hiddenCanvas.width, \_hiddenCanvas.height);
     20. // Transfer (and scale) the image from the hidden canvas to the main canvas
     21. \_mainContext.drawImage(\_hiddenCanvas, 0, 0, \_mainCanvas.width, \_mainCanvas.height);
     22. // Disable the Save button
     23. \_saveButton.setAttribute("disabled", "");

}

* 1. Press F5 to launch the application.
  2. Load a photo. Then put two fingers on the screen and move them apart to zoom in.
  3. Use a finger to reposition the photo so red eyes are roughly in the center of the screen.
  4. Tap a red eye in the photo to fix it (Figure 3).
     1. 
     2. Figure 3
     3. A zoomed photo with a red eye removed
  5. Put two fingers on the screen and move them together to zoom back out.
     1. **Note:** By default, scaling performed with CSS transforms occurs from the upper-left corner of a DOM element, causing the element to move unnaturally in response to pinch gestures. In Contoso Photo, the image scales outward from its center thanks to the “-ms-transform-origin: 50%” style applied to the image in default.css.
  6. Return to Visual Studio and stop debugging.

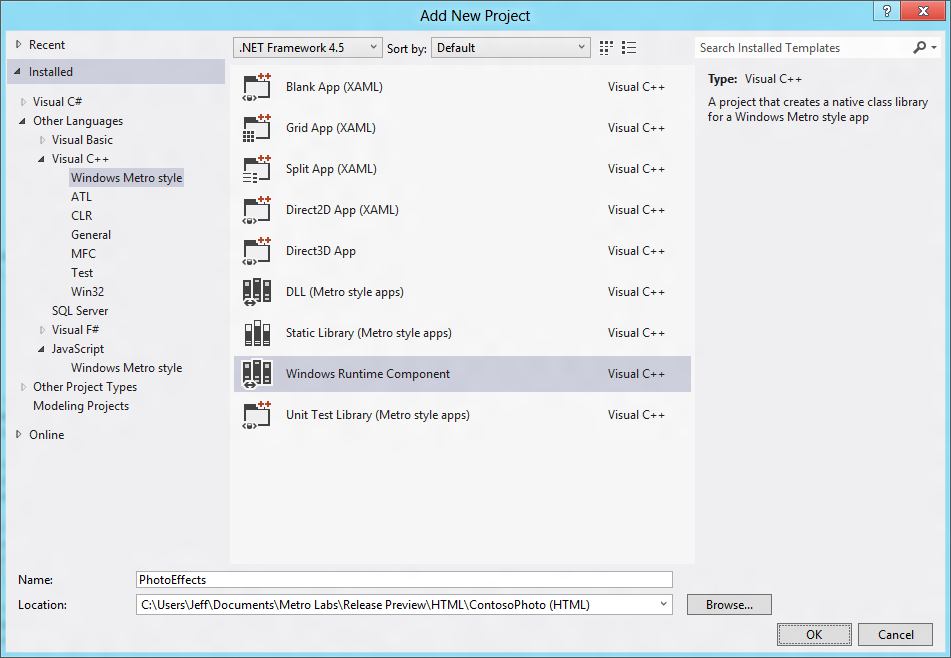
Task 4 – Add Mousewheel Support

* 1. Using MSGestureChange events for pinch-zooms works great on touch screens, but what about users who don’t have a touch screen? In this task, you’ll add mousewheel support so users can zoom in and out using the mouse.
  2. In default.js, add the following statement to the end of the app.onactivated handler:
     1. JavaScript
     2. document.addEventListener("mousewheel", onMouseWheelChanged);
  3. Now add the following event handler to default.js:
     1. JavaScript
     2. function onMouseWheelChanged(e) {
     3. if (\_bitmap != null && e.ctrlKey) {
     4. if (e.wheelDelta > 0) {
     5. // Zoom in
     6. if (\_scale < 4.0) {
     7. \_scale += 0.1;
     8. \_mainCanvas.style.msTransform = "scale(" + \_scale + ") translate(" + \_dx + "px," + \_dy + "px)";
     9. }
     10. }
     11. else {
     12. // Zoom out
     13. if (\_scale > 1.0) {
     14. \_scale -= 0.1;
     15. \_mainCanvas.style.msTransform = "scale(" + \_scale + ") translate(" + \_dx + "px," + \_dy + "px)";
     16. }
     17. }
     18. }
     19. }
     20. **Note:** This handler scales the photo using the same transform that the MSGestureChange handler uses. But this handler is called when the mousewheel is rolled forward or backward with the Ctrl key held down.
  4. Launch the application and load a photo.
  5. Roll the mousewheel forward and backward while holding down the Ctrl key. Verify that the photo zooms in and out the same way it does in response to pinch gestures.
  6. Confirm that you can also reposition the photo by dragging it with the mouse.
  7. Return to Visual Studio and stop debugging.

Exercise 3: Hybrid Apps

1. You may not have thought about it, but when you use WinRT classes in JavaScript, you’re actually using components that were written in another language (C++). What’s interesting is that you can build WinRT components of your own in C++ or C# and consume them from the language of your choice. In this exercise, you’ll see what it takes to implement a WinRT component in C++. Then you’ll remove the red-eye removal code that’s currently written in JavaScript and use the C++ component to remove red eyes from photos.

Task 1 – Write a WinRT Component in C++

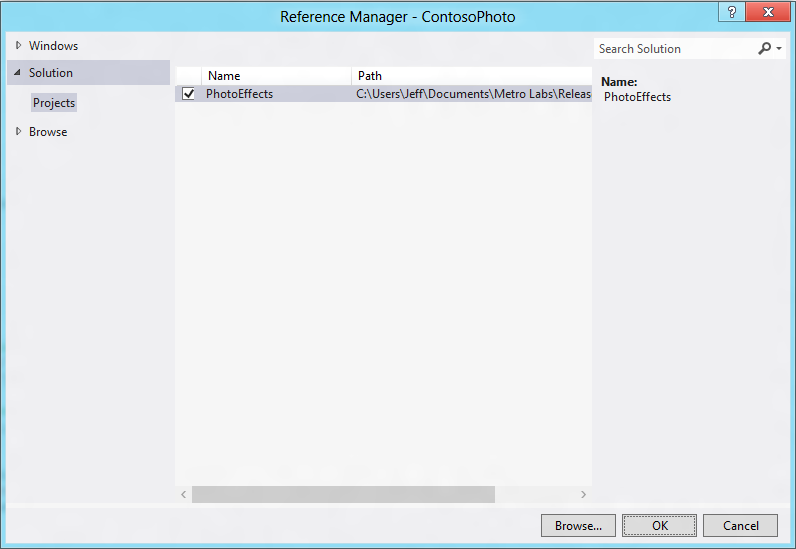
* 1. The first step in building a hybrid app is to build a component that can be called from JavaScript.
  2. Right-click the ContosoPhoto solution in Solution Explorer and use the **Add - New Project** command to add a Windows Runtime Component project named PhotoEffects to the solution (Figure 4). Be sure to select Visual C++ as the language.
     1. 
     2. Figure 4
     3. Adding a new C++ project to the solution
  3. In Solution Explorer, rename Class1.h to RedEyeComponent.h, and rename Class1.cpp to RedEyeComponent.cpp.
  4. Open RedEyeComponent.h and replace its contents with the following statements:
     1. C++
     2. #pragma once
     3. #include <collection.h>
     4. namespace PhotoEffects
     5. {
     6. public ref class RedEyeComponent sealed
     7. {
     8. public:
     9. RedEyeComponent();

Windows::Foundation::Collections::IVector<unsigned char>^ RemoveRedEye(Windows::Foundation::Collections::IVector<unsigned char>^);

};

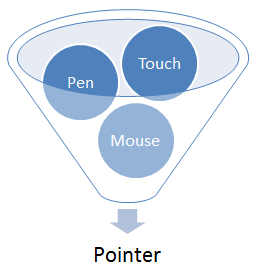
* + 1. }
  1. Open RedEyeComponent.cpp and replace its contents with the following statements:
     1. C++
     2. // RedEyeComponent.cpp
     3. #include "pch.h"
     4. #include "RedEyeComponent.h"
     5. using namespace PhotoEffects;
     6. using namespace Platform;
     7. using namespace Windows::Foundation::Collections;
     8. RedEyeComponent::RedEyeComponent()
     9. {
     10. }
     11. IVector<unsigned char>^ RedEyeComponent::RemoveRedEye(IVector<unsigned char>^ pixels)
     12. {
     13. // Loop through the pixels in the region, lowering red values that
     14. // exceed the sum of the blue and green values
     15. for (unsigned int i = 0; i < pixels->Size; i += 4)
     16. {
     17. unsigned char r = pixels->GetAt(i + 0);
     18. unsigned char g = pixels->GetAt(i + 1);
     19. unsigned char b = pixels->GetAt(i + 2);
     20. if (r > (g + b))
     21. {
     22. pixels->SetAt(i + 0, (unsigned char)((g + b) / 2));
     23. }
     24. }
     25. return pixels;
     26. }
  2. Right-click the PhotoEffects project in Solution Explorer and use the **Build** command to build the project. Verify that it builds without errors.

Task 2 – Call the WinRT Component from JavaScript

* 1. Now that the component is built, all you have to do to call it from JavaScript is create a reference to the C++ project in your JavaScript project, and then treat the C++ component like any other component. Here’s how.
  2. In Solution Explorer, right-click the References folder in the ContosoPhoto project and select the **Add Reference** command.
  3. Click Solution in the left pane to show the other projects in the solution. Then check the box next to “PhotoEffects” as shown in Figure 5, and click OK to add the project reference.
     1. 
     2. Figure 5
     3. Adding a reference to the C++ project
  4. Open default.js and remove (or comment out) the following statements from the removeRedEye function:
     1. JavaScript
     2. // Loop through the pixels in the region, lowering red values that
     3. // exceed the sum of the blue and green values
     4. for (var i = x1; i < x2; i++) {
     5. for (var j = y1; j < y2; j++) {
     6. var index = (i + j \* \_bitmap.width) << 2;
     7. var r = \_bitmap.data[index + 0];
     8. var g = \_bitmap.data[index + 1];
     9. var b = \_bitmap.data[index + 2];
     10. if (r > (g + b)) {
     11. \_bitmap.data[index + 0] = (g + b) / 2;
     12. }
     13. }
     14. }
  5. Replace the statements you just deleted with the ones below:
     1. JavaScript
     2. // Create a copy of the region
     3. var pixels = [];
     4. for (var i = x1; i < x2; i++) {
     5. for (var j = y1; j < y2; j++) {
     6. var index = (i + j \* \_bitmap.width) << 2;
     7. pixels.push(\_bitmap.data[index + 0]);
     8. pixels.push(\_bitmap.data[index + 1]);
     9. pixels.push(\_bitmap.data[index + 2]);
     10. pixels.push(\_bitmap.data[index + 3]);
     11. }
     12. }
     13. // Remove red eye from the region
     14. var effect = new PhotoEffects.RedEyeComponent();
     15. pixels = effect.removeRedEye(pixels);
     16. // Copy the modified pixels to the bitmap
     17. var k = 0;
     18. for (var i = x1; i < x2; i++) {
     19. for (var j = y1; j < y2; j++) {
     20. var index = (i + j \* \_bitmap.width) << 2;
     21. \_bitmap.data[index + 0] = pixels[k++];
     22. \_bitmap.data[index + 1] = pixels[k++];
     23. \_bitmap.data[index + 2] = pixels[k++];
     24. \_bitmap.data[index + 3] = pixels[k++];
     25. }
     26. }
  6. Press F5 to launch the application.
  7. Open a photo and tap a red eye to fix it. Verify that the red eye goes away just as it did before.
  8. Return to Visual Studio and stop debugging.

Exercise 4: Pointer Input

1. Windows 8 unifies all input so developers can code to one API that can handle touch, mouse, and pen input. The heart of this API is pointer events. In HTML, these events surface as DOM events with names such as MSPointerDown, MSPointerMove, and MSPointerUp. The beauty of pointer events is that they offer a unified and easy-to-use model for processing pointer input (see below). In addition, they fire for each pointer in contact with the screen, so you can use them to build rich multi-touch interfaces. And each time a pointer event fires, you can determine what type of pointer precipitated the event in case you wish to respond differently to different types of pointing devices.

**

1. In this exercise, you will add to Contoso Photo support for annotating images by drawing on them with a finger, the mouse, or a pen or stylus. You will use pointer events to process the input, and you will respond to these events by drawing lines on the HTML5 canvas that displays the photo.
   1. **Note:** this exercise is for demonstration purposes only. The figures you draw aren’t preserved if the app is suspended and terminated (although that’s a feature you could add as an exercise on your own), they don’t get saved when you save the image, and they go away if you tap or double-tap the image. Still, a lab on touch input wouldn’t be complete without a mention of pointer events!

Task 1 - Handle Pointer Events

* 1. The first task is to modify default.js to handle pointer events.
  2. Open default.js and add the following variable declarations to the other variable declarations at the top of the file:
     1. JavaScript
     2. var \_annotating = false;
     3. var \_redEyeButton;
     4. var \_annotateButton;
     5. var \_points = {};
  3. Add the following statement to the end of the app.onactivated handler:
     1. JavaScript
     2. // Register handlers for pointer events
     3. \_mainCanvas.addEventListener("MSPointerDown", onPointerDown);
     4. \_mainCanvas.addEventListener("MSPointerMove", onPointerMove);
     5. \_mainCanvas.addEventListener("MSPointerUp", onPointerUp);
     6. \_mainCanvas.addEventListener("MSPointerOut", onPointerOut);
  4. Add the following event handlers to default.js to track pointers as they move across the HTML5 canvas and draw a line connecting the current pointer position to the previous pointer position each time an MSPointerMove event fires:
     1. JavaScript
     2. function onPointerDown(e) {
     3. if (\_annotating && e.currentPoint.isInContact) {
     4. \_points[e.pointerId] = e.currentPoint;
     5. \_mainContext.lineWidth = 8;
     6. \_mainContext.lineCap = "round";
     7. \_mainContext.strokeStyle = "Yellow";
     8. }
     9. }
     10. function onPointerMove(e) {
     11. if (\_annotating && e.currentPoint.isInContact) {
     12. var point = e.currentPoint;
     13. if (\_points[e.pointerId] !== undefined) {
     14. var prev = \_points[e.pointerId];
     15. \_mainContext.beginPath();
     16. \_mainContext.moveTo(prev.position.x, prev.position.y);
     17. \_mainContext.lineTo(point.position.x, point.position.y);
     18. \_mainContext.stroke();
     19. }
     20. \_points[e.pointerId] = point;
     21. }
     22. }
     23. function onPointerUp(e) {
     24. if (\_annotating && \_points[e.pointerId] !== undefined) {
     25. delete \_points[e.pointerId];
     26. }
     27. }
     28. function onPointerOut(e) {
     29. if (\_annotating && \_points[e.pointerId] !== undefined) {
     30. delete \_points[e.pointerId];
     31. }
     32. }
     33. **Note:** To prevent pointer events used for drawing from conflicting with MSGesture events used for panning and zooming, we’ll only draw on the canvas when \_annotating is true. In the next task, you’ll add buttons that toggle \_annotating between true and false.
     34. Also note that instead of simply storing the previous pointer position so we can connect it to the current pointer position with a line, we’re storing previous pointer positions in array entries keyed by pointer IDs. This is essential if there are multiple pointers in contact with the screen so that the current position indicated by one pointer doesn’t get connected with the previous position associated with another pointer.
  5. Add the if condition to the onGestureChange function:
     1. JavaScript
     2. function onGestureChange(e) {
     3. if (!\_annotating) {
     4. \_dx += e.translationX;
     5. \_dy += e.translationY;
     6. \_scale = \_scale \* e.scale;
     7. \_scale = Math.min(\_scale, 4);
     8. \_scale = Math.max(\_scale, 1);
     9. e.target.style.msTransform = "scale(" + \_scale + ") translate(" + \_dx + "px," + \_dy + "px)";
     10. }
     11. }
  6. Make a similar change to the onImageTapped function:
     1. JavaScript
     2. function onImageTapped(e) {
     3. if (\_bitmap !== null && !\_annotating) {
     4. var x, y;
     5. // Get the click coordinates
     6. x = e.offsetX;
     7. y = e.offsetY;
     8. // Translate click coordinates into image coordinates
     9. x = Math.round(x \* \_hiddenCanvas.width / \_mainCanvas.width);
     10. y = Math.round(y \* \_hiddenCanvas.height / \_mainCanvas.height);
     11. // Fix red eye in region under cursor
     12. removeRedEye(x, y);
     13. // Enable the Save button
     14. \_saveButton.removeAttribute("disabled");
     15. }
     16. }

Task 2 – Modify the Application Bar

* 1. Next we need to add a couple of buttons to the application bar so the user can toggle between red-eye mode and annotate mode.
  2. In default.html, add the “reEyeButton” and the “annotateButton” statements to the DIV whose ID is “appbar”:
     1. HTML
     2. <div id="appbar" data-win-control="WinJS.UI.AppBar">
     3. <button id="openButton" data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'open', label:'Open', icon:'&#xE160;', section: 'selection'}"></button>
     4. <button id="saveButton" data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'save', label:'Save', icon:'&#xE105;', section: 'selection'}"></button>
     5. <button id="redEyeButton" data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'open', label:'Red Eye', icon:'&#xE15E;', section: 'global'}"></button>
     6. <button id="annotateButton" data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'save', label:'Annotate', icon:'&#xE104;', section: 'global'}"></button>
     7. </div>
  3. Open default.js and add the following statements to the app.onactivated handler, right after the statements that initialize the Open and Save buttons:
     1. JavaScript
     2. \_redEyeButton = document.querySelector("#redEyeButton");
     3. \_redEyeButton.addEventListener("click", onRedEyeButtonClicked);
     4. \_redEyeButton.setAttribute("disabled", "");
     5. \_annotateButton = document.querySelector("#annotateButton");
     6. \_annotateButton.addEventListener("click", onAnnotateButtonClicked);
  4. Add the following event handlers to default.js:
     1. JavaScript
     2. function onRedEyeButtonClicked(e) {
     3. \_annotating = false;
     4. \_redEyeButton.setAttribute("disabled", "");
     5. \_annotateButton.removeAttribute("disabled");
     6. }
     7. function onAnnotateButtonClicked(e) {
     8. \_annotating = true;
     9. \_redEyeButton.removeAttribute("disabled");
     10. \_annotateButton.setAttribute("disabled", "");
     11. }

Task 3 – Test the Results

* 1. Now let’s test the results and see pointer events in action.
  2. Press F5 to launch the application.
  3. Open a photo. Then display the application bar and select the Annotate button.
  4. Use a finger or the mouse to draw on the photo. Confirm that strokes appear in yellow, as shown in Figure 6.
     1. 
     2. Figure 6
     3. Annotating a photo
  5. If you’re running on a touch screen, draw with two or three fingers. Verify that you can draw with multiple fingers simultaneously.
  6. Use the mouse to draw on the photo. Can you draw with a mouse and a finger at the same time? Try it!
  7. Return to Visual Studio and stop debugging.

Summary

* 1. Great Metro style apps feature great touch interfaces. And Metro’s touch API, which is centered around MSPointer and MSGesture events, makes it easy to build great touch APIs. You saw that first-hand in this lab. You also saw how easy it is to call a WinRT component written in another language from JavaScript.
  2. As a parting thought, realize that you can do even more with touch than what we did in Contoso Photo. For example, in addition to including scaling information for implementing pinch zooms, MSGestureChange events include information specifying how much two fingers in contact with the screen have rotated relative to each other. If you wanted to allow the user to rotate a photo, you could add just a few lines of code that transfer rotation values to the CSS transform used for panning and zooming. With Windows 8 Metro, the only limit is your imagination.