Windows Bridge for iOS

[UI Touch Input]

DEV DESIGN specification

# [Overview](http://windowsblue/docs/home/Windows%20Spec%20Wiki/Dev%20Overview.aspx)

## [Summary](http://windowsblue/docs/home/Windows%20Spec%20Wiki/Dev%20Overview.aspx)

This document covers a fundamental pillar of WinObjC’s overall [UIKit strategy](https://microsoft.sharepoint.com/teams/Islandwood/_layouts/15/guestaccess.aspx?guestaccesstoken=vjdmNeR0we%2fCViDOOTJjKKRmIYrE2hR40hx9PFo2gn8%3d&docid=2_11aad47e362d24c4e93c54da26003f5cf&rev=1): the ***integration of WinObjC’s touch input handling with that of XAML and WinRT***.

As we begin to incorporate more XAML controls and features into our UIKit implementation, it is critical that we are able to provide a consistent hit-testing and touch input model to app developers in all circumstances, regardless of the type of UIControl/UIView that is receiving the touch input.

Currently, the XAML controls that are backing some of our UIKit controls (UITextField, etc.) act more like ‘XAML Islands’ than actual UIKit controls; they follow a completely isolated hit-testing and touch input processing path. This is unacceptable in many cases, as these ‘XAML Islands’ are unable to properly take part in the expected iOS model for touch input (hit-testing, UIResponder chain, UIGestureRecognizers, etc.).

In order to successfully build our UIKit controls and features on top of XAML and WinRT, while also providing a compelling UIKit porting solution to WinObjC app developers, we must merge the two touch input paradigms into a ***functional*** and ***consistent*** model that app developers can trust and build upon.

We will strive for ~100% app compatibility, but where that’s not possible (while also aligning with XAML/WinRT), we will identify and ***document patterns*** that app developers can use to ***work around the known discrepancies*** between the iOS and Windows platforms.

# [Design](http://windowsblue/docs/home/Windows%20Spec%20Wiki/Dev%20Overview.aspx)

## Overview

Before diving into our input handling strategy, we need to provide some background in the following areas:

### iOS Multitouch Behavior

The *high-level* path for processing [multi-touch events](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/multitouch_background/multitouch_background.html) on iOS is as-follows:

1. User touches or moves a touch point on the screen, thus creating or updating an existing [touch event](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/multitouch_background/multitouch_background.html).
   1. Each touch event transitions through various phases; UITouchPhaseBegan, UITouchPhaseMoved, UITouchPhaseEnded, and UITouchPhaseCancelled.
2. UIKit performs a [hit-test](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) to find the hit-test view.
3. The touch event is run through all [gesture recognizers](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizer_basics/GestureRecognizer_basics.html) corresponding to the hit-test view (starting at its utmost superview (usually its UIWindow), then all the way back down to the hit-test view).
   1. If a gesture was **NOT** recognized, the method corresponding to the touch’s phase is called on the hit-test view. For example; for UITouchPhaseBegan, [touchesBegan:withEvent:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIResponder_Class/index.html) is called on the hit-test view, passing it the new touch point.
   2. If a gesture **WAS** recognized, the touch’s phase is transitioned to UITouchPhaseCancelled, and [touchesCancelled::withEvent:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIResponder_Class/index.html) is fired to the hit-test view.
4. The touch event that is sent to the hit-test view in 3.a or 3.b above *may* be handled directly within the hit-test view, and it *may also* be sent up the [UIResponder](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) chain until it is handled elsewhere (or it is ultimately dropped by UIApplication). For example; touch events may go up the UIResponder chain to be handled by the hit-test view’s owning UIViewController, or even further up the UIResponder chain to be handled by a super UIView or its associated UIViewController, and so on.
   1. UITouches sent to UIControls *also* fire additional [UIControl events](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIControl_Class/) to any interested subscribers.

### WinObjC XAML Interop

As detailed in the [UIKit/XAML strategy doc](https://microsoft.sharepoint.com/teams/Islandwood/_layouts/15/guestaccess.aspx?guestaccesstoken=vjdmNeR0we%2fCViDOOTJjKKRmIYrE2hR40hx9PFo2gn8%3d&docid=2_11aad47e362d24c4e93c54da26003f5cf&rev=1), all WinObjC UIView instances are backed by a XAML Panel for layout, rendering, and input purposes. These XAML Panels are instances of the concrete (WinObjC-specific) CALayerXaml class.

At a high level, common patterns for each UIView’s CALayerXaml instances are as follows:

1. **Child CALayerXaml instances for each child UIView:**



1. **XAML Bitmap or XAML TextBlock for rendering a given UIView’s content:**



1. **Hosted XAML Controls:**



WinObjC’s current touch input processing implementation properly handles patterns #1 and #2 above, but it doesn’t accommodate for pattern #3. This means that every UIKit control that we implement via a backing XAML control relies ***solely*** on XAML for touch input processing, rather than integrating with the rest of the app’s UIResponder chain.

Without support for #3 above, many apps won’t port properly, and it often won’t be clear to developers why things aren’t working as expected.

***We must design a solution that integrates XAML control touch input into our UIKit touch input processing path***.

**Note:** Although much of the above architecture will likely change as we streamline our layout and composition architecture, the general approach to input handling that’s laid out in this document should be able to remain relatively unchanged. Specifically, we *may* be able to get rid of the backing CALayerXaml in #3 above, but arbitrary UIViews will still need *some* form of a backing XAML UIElement that we can leverage for touch input processing.

## Hit-testing

WinObjC’s UIKit implementation currently replicates the [iOS hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) model for all UIViews, and it functions well for that purpose.

However, as we continue to leverage more XAML controls for our UIKit control implementations, the fundamental difference between the [iOS hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) and [XAML hit-testing](https://msdn.microsoft.com/en-us/windows/uwp/xaml-platform/events-and-routed-events-overview) models becomes quite apparent; ***XAML does not provide the same level of flexibility for dynamically controlling UIElement hit-testing***. This results in a lack of consistency, which if not addressed, will be extremely challenging for app developers to overcome.

Fortunately, aside from the dynamic flexibility of the [iOS hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) model, the [iOS hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) and [XAML hit-testing](https://msdn.microsoft.com/en-us/windows/uwp/xaml-platform/events-and-routed-events-overview) models ***are actually quite similar***; both platforms are built upon the same general principles:

**Visibility**

* **XAML:** Only UIElements which are set to [Visibility=Visible](https://msdn.microsoft.com/library/windows/apps/br208992) are hit-testable.
* **iOS:** Only UIViews which are set to [hidden=NO](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) are hit-testable.

**Opaqueness**

* **XAML:** Only UIElements with a [non-null Background or Fill](https://msdn.microsoft.com/en-us/windows/uwp/xaml-platform/events-and-routed-events-overview) are hit-testable.
* **iOS:** Only UIViews with a [non-transparent alpha](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) are hit-testable.

**Explicit Hit-testability**

* **XAML:** Only UIElements which are set to [IsHitTestVisible=true](https://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.uielement.ishittestvisible) are hit-testable.
* **iOS:** Only UIViews which are set to [userInteractionEnabled=YES](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) are hit-testable.

### XAML Hit-testing Mapping

Given the similarities between the two platforms, ***we were able to perform a simple mapping of hit-testability properties between UIKit and XAML***, by modifying properties on each UIView’s backing CALayerXaml Panel as required.

The hit-testability mapping is as follows:

|  |  |
| --- | --- |
| **UIView Instance** | **Backing CALayerXaml** |
| userInteractionEnabled = NO/YES | IsHitTestVisible = false/true |
| hidden = YES/NO | IsHitTestVisible = false/true |
| alpha <= 0.1f / > 0.1f | IsHitTestVisible = false/true |

With the above mapping, we were able to remove all custom code for replicating the [iOS hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) model, thus ***aligning 100% with XAML’s hit-testing methodology.*** Additionally, this mapping has allowed us to achieve full app compatibility with all of our internal test applications, resulting in the need for ***zero internal test app code changes*** to date.

**Note:** The above mapping *augments* (rather than replaces) our preexisting support for UIView visibility and alpha states, which are still handled within our lower-level QuartzCore/XAML composition translation layer.

### Hit-testing Discrepancies

Although we achieved 100% internal app compatibility with the above changes, we have identified two primary discrepancies between the [XAML hit-testing](https://msdn.microsoft.com/en-us/windows/uwp/xaml-platform/events-and-routed-events-overview) and [iOS hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/event_delivery_responder_chain/event_delivery_responder_chain.html) models:

1. On iOS, [UIViews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) are able to dynamically take part in the hit-testing process, whereas XAML UIElements are not able to do so. This means that UIViews can dynamically decide whether or not they want to accept touch input ***even while the hit-test is occurring***. The Apple documentation explicitly recommends this technique for intercepting touches in subviews (see [overriding hit-testing](https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/multitouch_background/multitouch_background.html)), but we haven’t seen this method used in any of our internal test applications.

This differs from XAML, where UIElements are either hit-testable or not, with ***no way to override/augment the XAML hit-test path***.

1. On iOS, any point in a subview that is outside the bounds of its superview can’t receive touch events, because the touch point has to be within the bounds of the superview ***and*** the subview for it to receive touch input. This can occur if a subview’s [clipsToBounds](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/index.html) property is set to NO, while also exceeding the bounds of its superview.

This differs from XAML, where the *entire* child UIElement is either hit-testable or not; there’s no clipping of its hit-test region based upon its parent’s dimensions. For example:



We haven’t seen these patterns used in any internal apps, but it is ***very likely*** that they will be used by external app developers who are interested in WinObjC. As a mitigation for such discrepancies, ***app developers should* *leverage the userInteractionEnabled property on their UIViews, and/or add additional UIViews*** for controlling hit-testing, rather than relying on behaviors #1 and #2 above.

Hit-test troubleshooting techniques are outlined in the next section.

### Troubleshooting Hit-testing Discrepancies

#### HIT-TEST Logging

Extensive logging has been added to UIView to aid in hit-test debugging (this tracing helped immensely while making these hit-testing changes). By default, debug builds of the WinObjC SDK will trace any flagged hit-testing discrepancies to the debugger.

Such tracing will appear as follows:

W/UIView: XAML's chosen hit test view:

UITableViewContentView(0x12279cec, \_isHitTestable=true, XAML\_HitTestable=true)<-UITableViewCell(0x121b97f4, \_isHitTestable=true, XAML\_HitTestable=true)<-UITableView(0x0e94de3c, \_isHitTestable=true, XAML\_HitTestable=true)<-UIEmptyController(0x1227759c, \_isHitTestable=true, XAML\_HitTestable=true)<-UIView(0x0e9a7164, \_isHitTestable=true, XAML\_HitTestable=true)<-UINavigationPane(0x0e9a8044, \_isHitTestable=true, XAML\_HitTestable=true)<-UIWindow(0x0796528c, \_isHitTestable=true, XAML\_HitTestable=true)

...does not match the legacy hit test results:

UITableViewCell(0x121b97f4, \_isHitTestable=true, XAML\_HitTestable=true)<-UITableView(0x0e94de3c, \_isHitTestable=true, XAML\_HitTestable=true)<-UIEmptyController(0x1227759c, \_isHitTestable=true, XAML\_HitTestable=true)<-UIView(0x0e9a7164, \_isHitTestable=true, XAML\_HitTestable=true)<-UINavigationPane(0x0e9a8044, \_isHitTestable=true, XAML\_HitTestable=true)<-UIWindow(0x0796528c, \_isHitTestable=true, XAML\_HitTestable=true)

You may recompile the WinObjC SDK if further hit-testing details are desired. First, alter UIView.mm by setting DEBUG\_HIT\_TESTING to true, then recompile.

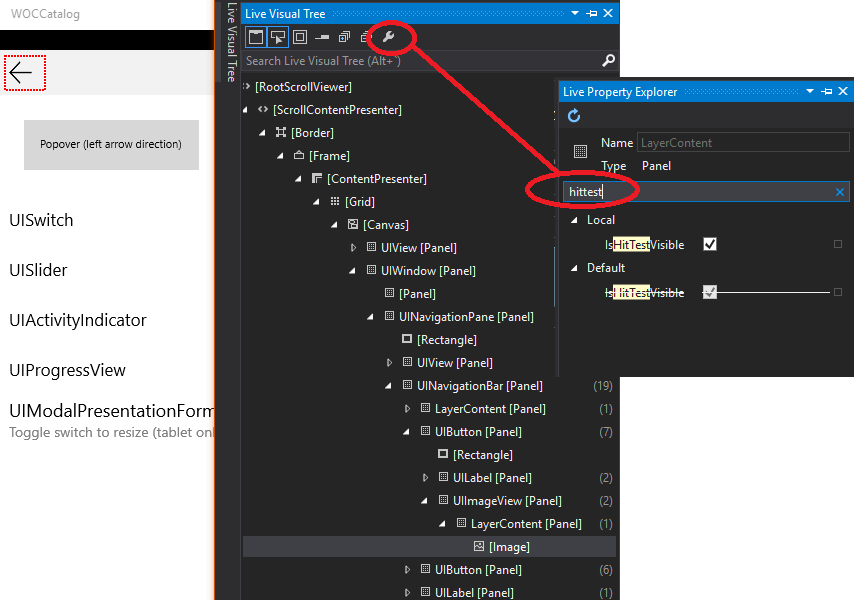
**Note:** the default setting is currently DEBUG\_HIT\_TESTING\_LIGHT:

static const bool DEBUG\_HIT\_TESTING = DEBUG\_ALL || false;

static const bool DEBUG\_HIT\_TESTING\_LIGHT = DEBUG\_HIT\_TESTING || true;

#### Visual Studio’s Live Visual Tree Viewer

We have added user-friendly names to all of our UIElements in the XAML tree, so app developers can leverage Visual Studio’s Live Visual Tree viewer to troubleshoot hit-test and layout issues:

****

**Note:** Curiously, the actual CALayerXaml panels (for each UIView, UIWindow, etc.) in the UIElement tree above are only 1x1, so be sure to use their contents (Rectangles, Images, etc.) to troubleshoot hit-testing rather than looking solely at the 1x1 panels themselves. The need for these intermediate 1x1 panels will be resolved in a subsequent refactor.

#### Legacy Hit-testing Behavior

#### We’ve added a *temporary* option exposed off of WOCDisplayMode which allows developers to favor the legacy hit testing model if a hit testing discrepancy is found. Although not 100% backward compatible, we’ll provide it as an escape hatch to unblock developers as we roll out the new hit-testing model.

#### A sample has been added to WOCCatalog (under DisplayMode) that demonstrates how to dynamically switch between the old/new hit testing models:

UIApplication.displayMode.useLegacyHitTesting = YES;

#### Note: This feature is for transitional purposes only and *will be deprecated in the near future*.

## Touch Handling

WinObjC touch handling falls into two distinct buckets;

* [UIView Touch Handling](#_UIView_Touch_Handling)
* [XAML Control Touch Interop](#_XAML_Control_Touch)

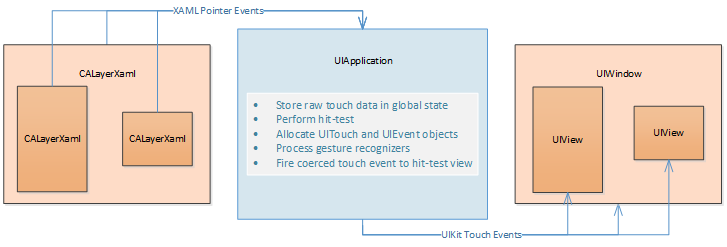
### UIView Touch Handling

As outlined above, all UIViews in WinObjC are backed by a XAML Panel (aka CALayerXaml) instance for layout, rendering, *and* *input* purposes.

WinObjC’s legacy touch handling solution hooked each CALayerXaml instance’s pointer events from within our QuartzCore/XAML composition translation layer, and fed them *all the way out* to the global UIApplication instance for processing.

UIApplication's legacy processing included:

1. Storing touch data in global state, including tracking pointer tap counts and velocity.
2. Performing hit-testing.
3. Storing/updating the current touch points within each view.
4. Allocating/populating new UITouch and UIEvent objects per XAML pointer event.
5. Processing gesture recognizers on the hit-test view and all of its superviews.
6. Firing each gesture-recognizer-coerced touch event to the hit-test view.

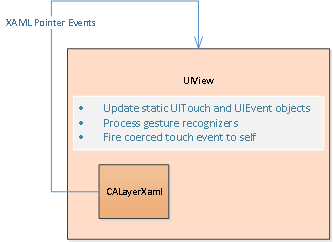


Such processing of pointer events at the global UIApplication level required UIApplication to manage the relationships between, and the intimate inner details of, UIView, UIGestureRecognizer and UITouch states.

WinObjC's new touch input processing model separates touch-handling logic into the various components that are ultimately responsible for them, resulting in a simplified, componentized, and more approachable design for us to build upon.

Some key pieces of the new design include:

1. Hit-testing is now performed by XAML.
2. Hit-tested UIViews receive their pointer events directly from XAML, and internally translate them to UITouch events.
3. Static UITouch and UIEvent instances are owned within UIView, and they are updated as-needed as XAML pointer events are fired.
4. UITouch instances now track and calculate *their own* tap counts and velocity.
5. Each UIView manages *its own* internal list of current touch points.
6. UIView now contains the gesture-recognizer-processing logic.
   1. Note that this should *eventually* be moved into UIGestureRecognizer as called out below.
7. UIView instances fire the final gesture-recognizer-coerced UITouch events *to themselves* (indeed, they’re the hit-test view).



This new input processing model requires hit-tested UIView instances to ***receive XAML pointer events directly from their backing CALayerXaml instances***. In order to do so, UIView now owns a reference to its backing CALayerXaml instance (held as a WinRT-projected XAML FrameworkElement, aka WXFrameworkElement\*), which we refer to as its ***XamlInputElement***.

Upon initialization, UIView subscribes to its ***XamlInputElement's*** relevant pointer events, and provides event handler blocks which process pointer events for their corresponding UITouchPhase.

The XAML-pointer-to-UITouchPhase mapping is as follows:

|  |  |  |
| --- | --- | --- |
| **XAML Pointer Event** | **UITouch Event** | **UITouchPhase** |
| OnPointerPressed | touchesBegan:withEvent: | UITouchPhaseBegan |
| OnPointerMoved | touchesMoved:withEvent: | UITouchPhaseMoved |
| OnPointerReleased | touchesEnded:withEvent: | UITouchPhaseEnded |
| OnPointerCanceled/OnPointerCaptureLost (duplicate/unexpected calls are dropped) | touchesCanceled:withEvent: | UITouchPhaseCanceled |

Within each of the ***XamlInputElement’s*** OnPointer\* event handling block, the incoming XAML WUXIPointerRoutedEventArgs\* instances are set to ‘handled’ (so they don't bubble out to parent XAML elements), and are then passed into UIView's internal \_processPointerEvent:forTouchPhase: method, where they are processed as UITouches and dispatched to the ported application.

#### UIView Pointer Events

**\_processPointerEvent:forTouchPhase:** is responsible for the following tasks:

1. Retrieve the WUIPointerPoint from the WUXIPointerRoutedEventArgs
2. Convert the WUIPointerPoint to the coordinate space of the UIView's parent UIWindow (the coordinate space of the UIWindow’s backing CALayerXaml panel).
3. Begin tracking global state for the UITouch instance that is tied to that WUIPointerPoint Id.
4. If the pointer event corresponds to UITouchPhaseBegan, capture the WUIPointerPointer.
5. Perform a legacy hit test (if enabled), and log any flagged discrepancies to the debugger (to aid in hit-test debugging).
   1. This is also where the app developer is able to toggle the old hit-testing behavior (although that functionality will be deprecated in the near future).
6. Run the UITouch event through all gesture recognizers that are attached to the hit-test view, or any of its superviews.
7. If the gesture was recognized, transition the UITouch event's phase to UITouchPhaseCancelled.
8. Update the hit-test UIView's internal list of current touches as required for the given UITouchPhase.
9. Send the UITouch event to the hit-test UIView.
   1. Note that this event may have transitioned to UITouchPhaseCancelled in response to a gesture being recognized (#7 above).
10. Return the gesture-recognizer-coerced UITouchPhase back to the caller.

### XAML Control Touch Interop

Our default UIView touch handling requires pointer events to fire on each hit-tested UIView’s backing CALayerXaml instance, allowing us to process and translate them to the UITouch events that are required by ported applications.

This poses a problem for any of our UIKit control implementations that are backed by XAML controls, because such XAML controls span the *entire* dimensions of their backing CALayerXaml panels. For example, the CALayerXaml instance that contains the below XAML Button won’t *ever* receive any pointer events, because they are sent to, and are handled by, the hosted XAML Button:



The implications of this are that any pointer events that go directly to the XAML controls that we are leveraging won’t be able to take part in the applications’ UITouch input processing or gesture recognition logic. Applications use a number of patterns that are built upon proper UITouch event notifications, including:

* Subclassing UIKit controls, in order to receive their [touchesBegan:withEvent:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIResponder_Class/), etc. function calls.
* Swizzling UIKit control methods, in order to receive their [touchesBegan:withEvent:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIResponder_Class/), etc. function calls.
* Subscribing to UIControl notification events via [addTarget:action:forControlEvents:.](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIControl_Class/)
  + For example, it’s common to subscribe to [UIControlEventTouchUpInside](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIControl_Class/) on a UIButton, to be notified when it’s been clicked.
* Attaching UIGestureRecognizers to UIViews either directly on the UIKit control, or on any of its superviews.

***It’s critical that we provide all of this functionality on our platform***. In order to do so, we must intercept each adapted XAML control’s pointer events, so we can process and dispatch them as the app intended. XAML provides two mechanisms for intercepting UIElement pointer input:

#### XAML CONTROL Subclassing

Ideally, we could intercept pointer events on our XAML controls ***as they occur***, so we could feed them through any registered UIGestureRecognizers, and *then* forward them into the touched XAML control *only if a gesture wasn’t recognized*.

XAML’s solution for *pointer interception* is to subclass a given XAML control, and to override its OnPointer\* methods. However, that leads to a few complications:

1. **Sealed Controls:** Many of the XAML controls that we need to adapt are sealed, so subclassing isn’t even an option in those cases.
2. **Composited Controls:** XAML controls are often built by composing various other XAML controls together, and in such cases, we’d need to subclass *all of them* if we want to be able to process their pointer events as UITouch input. Furthermore, some XAML controls may be unsealed, but it’s common that they are *composed* of controls that are sealed, which leads us back to issue #1 above. For example, XAML’s Slider control is unsealed, but its *internal* Thumb is not, so we are unable to intercept any pointer events that are consumed by the Slider’s internal Thumb control.

***Given the complexities and limitations with XAML subclassing, we are going to avoid it for our pointer-processing needs, and instead rely on XAML control pointer events.***

#### XAML CONTROL Pointer Events

The second method for receiving pointer events for XAML controls involves event subscription ([PointerPressed](https://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.uielement.pointerpressed), etc.). It’s critical to note that these events are fired ***after*** the XAML control has already received and handled (or chosen not to handle) them. This means that if a XAML control that we’re using performs its own pointer processing (and most of them do), we can’t use these bubbled pointers events to modify or prevent that control’s default pointer event handling, but ***it looks as though this will suffice for our needs***.

Additionally, when a XAML control handles a given pointer event, it *does not* bubble out to subscribers. For example; *we will never receive* them by subscribing directly to Button’s [PointerPressed](https://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.uielement.pointerpressed) event, because Button *always* handles/consumes them.

In order to receive a control’s *handled* pointer events, we must instead subscribe to them via UIElement’s [AddHandler](https://msdn.microsoft.com/en-us/library/windows/apps/hh702399.aspx) method, passing ‘true’, so we can receive *both* handled *and* unhandled events on the control.

***This is the method that we will use for dispatching XAML controls’ pointer events as UITouch input to WinObjC applications.***

Each UIKit control that adapts a XAML control will need to add event handlers for the following events: PointerPressedEvent, PointerMovedEvent, PointerReleasedEvent, PointerCanceledEvent, and PointerCaptureLostEvent.

Handling of such events will be performed as follows:

1. Call into their base UIView’s \_processPointerEvent:forTouchPhase: method, passing the pointer event argument and the UITouchPhase that it maps to (PointerPressed=UITouchPhaseBegan, etc.).
2. If \_processPointerEvent:forTouchPhase: returns UITouchPhaseCancelled for a UITouchPhaseBegan or UITouchPhaseMoved event (meaning that a gesture was recognized), call [ReleasePointerCapture](https://msdn.microsoft.com/en-us/library/windows/apps/br208977.aspx) on the XAML control, to cancel the touch sequence on the XAML control. For example, this will prevent a XAML Button from firing its ‘clicked’ event if a UIGestureRecognizer has intervened.
3. Set the pointer event to ‘handled’ (if it is not already set), so it doesn’t bubble up further in the XAML UIElement tree.

**Forced Pointer Capture Release:** Is this supported usage of the [ReleasePointerCapture](https://msdn.microsoft.com/en-us/library/windows/apps/br208977.aspx) API for canceling a touch sequence on a XAML control? The goal is to prevent the XAML control from completing its touch sequence. For example; we want to prevent a XAML Button from firing its ‘clicked’ event if a gesture is recognized by any of the application’s UIGestureRecognizers. Will [ReleasePointerCapture](https://msdn.microsoft.com/en-us/library/windows/apps/br208977.aspx) work for this? It appears to work fine, but we need to confirm that there aren’t any gotchas here.

### Global Touch Input

**UIApplication::beginIgnoringInteractionEvents** is UIApplication’s mechanism to temporarily disable input for the entire application. WinObjC’s legacy implementation relied on being able to drop input events *at the UIApplication level*, but that is not possible with the new touch input design. Instead, the new implementation leverages CoreWindow’s IsInputEnabled property to disable all input for the UIApplication’s backing CoreWindow, until **UIApplication::endIgnoringInteractionEvents** is called.

## Gestures

### WinObjC contains the following hand-rolled UIGestureRecognizers:

* UILongPressGestureRecognizer
* UIPanGestureRecognizer
* UIPinchGestureRecognizer
* UISwipeGestureRecognizer
* UITapGestureRecognizer

These seem to work well for our internal test apps, and they also have continued to function properly after our touch input refactor. However, there are several arguments for rewriting them on top of WinRT GestureRecognizers:

1. UIView has a hard-coded prioritized list of UIGestureRecognizers in an attempt for a reasonable user experience; this seems a bit fragile:

// gesture priority list

const static id s\_gesturesPriority[] = {[UIPinchGestureRecognizer class],

[UISwipeGestureRecognizer class],

[UIPanGestureRecognizer class],

[UILongPressGestureRecognizer class],

[UITapGestureRecognizer class] };

1. Limited (or no?) support for *custom* UIGestureRecognizers (recently reported on GitHub)
2. UIGestureRecognizers track internal and global state, trying to cooperatively provide a reasonable user experience. For example, the UITapGestureRecognizer won’t fire if other UITapGestureRecognizers are waiting for more taps to occur.
3. We shouldn’t be maintaining gestures that already exist in WinRT.

Porting each of our UIGestureRecognizers to WinRT appears to be do-able, but it also looks like we can continue to function as-is in the near-term. When we do start porting our gesture recognizers, we will need to take the following into account:

1. Although XAML UIElements do expose some gestures, ***they are only for a single touch point, whereas iOS UIGestureRecognizers all have built-in multi-touch support***.
2. Due to #1 above, we’ll likely need to write our own UIGestureRecognizers that contain one WinRT GestureRecognizer *per touch point*, because WinRT GestureRecognizers are also only single-touch.
3. The input refactor described in this document already stores the PointerPoint on the UITouch instance, so UIGestureRecognizers that we port to WinRT will be able to easily receive the UITouch, grab its stored PointerPoint instance, and hand it off to its corresponding internal WinRT GestureRecognizer for processing.
4. Our current gesture recognition logic still lives in UIView, but it should really go into the UIGestureRecognizer base class.

### Unsupported UIGesture Features

There are a few UIGestureRecognizer features that WinObjC doesn’t currently support:

**Delayed Input (P2)**

UIGestureRecognizer’s [delaysTouchesBegan](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIGestureRecognizer_Class/#//apple_ref/occ/instp/UIGestureRecognizer/delaysTouchesBegan) and [delaysTouchesEnded](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIGestureRecognizer_Class/#//apple_ref/occ/instp/UIGestureRecognizer/delaysTouchesEnded) properties allow for delayed input to the hit-tested UIView, depending on whether or not any attached UIGestureRecognizers are potentially recognizing a gesture. This is *somewhat* similar to DManip gestures in ScrollView on Windows, but we don’t have any way to hook into the input stack to explicitly control this behavior, so ***it’s unlikely that WinObjC will ever support delayed UIGestureRecognizer input***.

**Gesture Dependencies (P2)**

iOS allows app developers to declare dependencies between their UIGestureRecognizers via the [requireGestureRecognizerToFail](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIGestureRecognizer_Class/#//apple_ref/occ/instm/UIGestureRecognizer/requireGestureRecognizerToFail:) method. This feature is not implemented in WinObjC today, but it looks as though there is nothing fundamentally blocking us from implementing the feature if/when needed.

## Rollout/Flighting

This is a large change that may impact our existing consumers. In order to prevent/minimize regressions in their apps, we will push a topic branch to GitHub, along with a blog post and this document, to gather early feedback from developers. At that point, we will shoot for ~1 week of bake time before pushing these changes to GitHub master.