Non-Linear Navigation Service for Windows Phone 7

A Windows Phone Recipe

# Purpose of This Document

This document introduces the Non-Linear Navigation Service (NLNS) for Windows Phone. It explains the reasons for creating the NLNS and the considerations that went into its design, how to use the NLNS in your Windows Phone Silverlight application, and provides a short review of its implementation.

# Why Do We Need the Non-Linear Navigation Service?

The Windows Phone execution model provides users with a consistent navigation experience in both native (built-in) and non-native applications. Native applications include things like the camera and the People Hub, while non-native applications include all third-party applications such as yours.

All Windows Phone devices feature a Back button (simply Back from here forward). This button allows the user to navigate back within a given application, between pages, and even between different applications. This is a rather cool feature, mainly because unlike with other phones, Windows Phone users can navigate from your application to other applications like the browser, maps, or the contact list, and then press Back to seamlessly return to your application. This yields a much more consistent user experience across different applications, whether they are third-party applications or part of the phone's built-in application suite.

On Windows Phone, users can launch an application by tapping the application tile on the phone’s Start screen, or by tapping the application icon from the installed applications list. Users can also navigate backwards through the pages of a running application or through the stack of previously running applications by pressing Back. The stack of previously running applications is known as the application back-stack.

The Windows Phone Silverlight application model uses pages as the building block of Silverlight applications. Most applications are built from multiple pages hosted in a frame, and the user has the ability to navigate between the application’s pages. For each application, the phone maintains a page back-stack that reflects the history of pages visited by the user.

Applications use page URIs to navigate to any of their pages. The URI (uniform resource identifier) provides the relative position of the XAML page in the XAP file. As pages are navigated to, the operating system builds up a stack in order to enable back navigation. This default behavior works great with most applications, but in applications with a complex page structure, the default navigation model creates user experience (UX) confusion and potential bugs.

## Silverlight Navigation for Windows Phone 101

While this article doesn’t provide a detailed explanation of the Silverlight Navigation for Windows Phone (see more details [here](http://msdn.microsoft.com/en-us/library/ff941091(v=VS.92).aspx)), we need to provide just enough information to explain the problem domain. Generally speaking, Windows Phone Silverlight applications use two types of navigation: **New** for when you navigate to a new page, or **Back** for when you navigate to the previous page.

### Windows Phone Navigation Services

Each Windows Phone Silverlight application is constructed of a frame (**PhoneApplicationFrame**) that contains a number of pages (**PhoneApplicationPage**). As a developer, you can use the navigation services that are built into each page and are governed by the application **RootFrame**. For example, the following code snippet is a simple navigation request from the current page - let’s assume Page1.xaml, to a new page called Page2.xaml

NavigationService.Navigate(new Uri("/Page2.xaml", UriKind.Relative));

As you use the **Navigate()** method, the phone keeps track of each move and creates the application’s page back-stack. You can also use the **NavigationServices.GoBack()** method, which navigates back to the previous page and pops the top page from the application’s page back-stack into view.

Pressing the phone’s Back button, yields the same effect as calling the **NavigationServices.GoBack()** method. In addition, pressing Back from the application’s first page exits the application. This is enforced by the Windows Phone Marketplace policy, since pressing Back on the application’s first page is the only way to exit the application.

### Page Navigation Example

The following navigation example summarizes everything you need to know about navigation in regard to the NonLinearNavigationService. The example includes 2 pages, Page#1 and Page#2. The following code snippet is the code behind for page #1:

// Constructor

public MainPage()

{

InitializeComponent();

Debug.WriteLine("In Main Page CTOR \_PageCount = " + \_PageCount);

\_PageCount++;

}

protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)

{

base.OnNavigatedTo(e);

int pageCount=0;

if (this.NavigationContext.QueryString.ContainsKey("PageCount"))

{

int.TryParse(this.NavigationContext.QueryString["PageCount"], out pageCount);

}

this.txtPageCount.Text = "Page count = " + pageCount.ToString();

}

// Fragmented navigation is not supported.

private void btnMoveNext\_Click(object sender, RoutedEventArgs e)

{

NavigationService.Navigate(new Uri("/Page1.xaml", UriKind.RelativeOrAbsolute));

}

// This creates a new instance of the page

private void btnMoveNextNew\_Click(object sender, RoutedEventArgs e)

{

NavigationService.Navigate(new Uri("/Page1.xaml?PageCount=" + \_PageCount.ToString(), UriKind.RelativeOrAbsolute));

}

private void btnMoveToPage2\_Click(object sender, RoutedEventArgs e)

{

NavigationService.Navigate(new Uri("/Page2.xaml", UriKind.RelativeOrAbsolute));

}

Here you can see that Page #1 has three buttons; all use the **Navigate()** method to go to different URIs:

* “Go To Main Page (again)” is the URI for the main page itself; if you press this button, you will learn that fragmented navigation is not allowed
* The “Go To Main Page (new)” URI also points to the main page, but adds a query parameter that increments each time the page is constructed; the increment occurs in the page constructor
* “Go To Page #2“ is the URI for page #2

Page#2 has only one button that uses the **Navigate()** method to navigate to Page#1. The URI is the first page URI (Page1.xaml).

* Run the application and tap on the first button, “Go To Main Page (again)”  
  This will raise an exception, as fragmented navigation is not supported by Windows Phone.
* Run the application and tap on the second button, “Go To Main Page (new)”  
  This will result in the application navigating to a **NEW** instance of the first page – Page #1. You can see that the page count increases each and every time you tap the “Go To Main Page (new)” button. Press Back to return to the previous page and you will see the page count decrease. Press Back several times until the page count returns to 0. The next tap on Back will cause the application to exit. You need to understand that each time you press the “Go To Main Page (new)” button, a completely **NEW** instance of the page is created, and is placed into the application’s page back-stack as illustrated by the following table:

|  |  |
| --- | --- |
| Page1 | Application’s default page |
| Page1?PageCount=1 | Navigating to a new Page instance |
| Page1?PageCount=2 | Navigating to a new Page instance |
| ….. | ….. |
| Page1?PageCount=N | Navigating to a new Page instance |

* Now run the application and tap on the last button, “Go To Page #2”  
  This will result in the application navigating to a NEW instance of the second page, Page #2.
* On Page #2, press “Navigate to Page #1”.   
  This results in the application navigating to a NEW instance of Page #1.
* From Page #1 press “Go To Page #2” and repeat this process several times.
* Now start pressing Back; you will notice that you are traveling back between the two pages several times before you exit the application, as illustrated by the following table:

|  |  |
| --- | --- |
| Page1 | Application’s default page |
| Page2 | Navigating to a new Page 2 instance |
| Page1 | Navigating to a new Page 1 instance |
| Page2 | Navigating to a new Page 2 instance |
| Page1 | Navigating to a new Page 1 instance |
| Page2 | Navigating to a new Page 2 instance |
| Page1 | Navigating to a new Page 1 instance |

From these tests, we learned that the Windows Phone Silverlight navigation model:

* Creates a new instance of the same “page” if you use the **NavigationServices.Navigate()** method
  + With the exception that the Windows Phone Silverlight navigation model doesn’t support fragmented navigation
* The application’s page back-stack grows as you use **NavigationServices.Navigate()**, including “duplication” of the “same pages”
  + Using only the **Navigate()** method to navigate between page will enlarge the application’s page back-stack to the point where the application might run out of memory
* The application’s page back-stack shrinks as you navigate back using the phone’s Back button
* “Navigation Loops” are defined as having multiple instances of the same page with the same URI – as shown in the last example

The important thing to remember is that the application’s page back-stack can include “duplicated” pages. Therefore, we can have many instances of Page #1 or Page #2 in the back-stack even if the URI used to navigate to these pages was identical for all Page #1 or Page #2 pages.

## The Non-Linear Navigation Problem

By understanding how Windows Phone Silverlight application navigation works, we can define the problem we are trying to solve. We want to eliminate any duplication of page instances in the application’s page back-stack as shown in the above examples. We want to avoid any Navigation Loops.

*The* ***UX guidelines*** *for Windows Phone are to* ***avoid*** *having “home” functionality in your application since it is the number one cause of navigation loops. But even then, you might end up with navigation loops that you need to handle.*

While the above examples are fake scenarios, they reflect a pattern in which non-linear navigation can create loops and circular navigation. This can result in user confusion and misleading navigation, where the user expects a particular navigation behavior from pressing Back, while in fact, due to the navigation loop, Back behaves differently.

For example: Assume a given application has four pages, the main page and pages 2, 3, and 4, where pages 2, 3, and 4 are part of a wizard for adding new items, and the main page presents a list of items.

* Upon application launch, the main page shows the list of items  
  Pressing Back at this stage cause the application to exit.
* Adding a new item, takes the user through the wizard to pages 2, 3, and 4 after which the application returns to the main page showing the list of items including the newly added item
* Using the Navigate() method creates a loop since now we have two main pages in the application’s page back-stack
  + What should happen when the user presses Back?
    - The user might expect the application to exit
    - But if we have a navigation loop, the application will return to the last page of the wizard – which is the wrong behavior

Another example: Assume a given application has a Home button on each of the application’s pages that enables the user to return to the application home page. Using the **Navigation()** method, to navigate home creates a navigation loop that the user might find odd.

Since the developer can’t manipulate the application’s page back-stack, once a loop is created, the loop can’t be removed other than by navigating back by using the Back button or the **NavigationServices.GoBack()** method.

# The Solution: Non-Linear Navigation Service

The recommended solution to the navigation loop problem defined above is to avoid navigation loops to begin with. This can be achieved by thoughtful UX design. However, while this could be appropriate for many applications, it imposes a severe system limitation on developers.

A more refined solution to the problem is to perform a recursive back navigation. This recursive back navigation will automatically navigate back through the pages found in the application’s page back-stack, until the beginning of the navigation loop. For this to work, you need to do the following:

1. Keep track of your navigation history in order to identify potential loops.
2. Once you identify a potential loop, start a recursive back by raising a “recursive back flag” and calling the NavigationServices.GoBack() method.
3. For each page in your application, in the OnNavigatedTo method, check if the “recursive back flag” is set; if so navigate back once more.

Since you can’t manipulate the application’s page back-stack, you are forced to perform the back navigation to clear the application’s page back-stack.

Luckily, we’ve solved that problem for you. Once it is initialized, the NonLinearNavigationService monitors the navigation in your application. When it detects a navigation loop (navigation to a page with a URI that already exists in the application’s page back-stack), it automatically performs the recursive back navigation for you. That is right, you need not do any additional work and this service will navigate back for you through the application’s page back-stack. Simply initialize the service in your application constructor using the following code snippet:

// init the non-linear nav service, make sure you call it last in the app CTOR

NonLinearNavigationService.Instance.Init(RootFrame);

From this point on, just keep on using the **Navigate**() method and the service will take care of everything for you.

## Optimizing the Recursive Back Navigation Experience

While the service takes care of everything for you, there are some optimizations that your application can perform to enhance the user experience. The reason you need such optimization lays in the Windows Phone Silverlight application navigation model, which forces the NLNS to navigate back through all the pages in the application’s page back-stack to the beginning of the navigation loop. When your pages include a large number of UI elements with complex data binding that gets loaded in the OnNavigatedTo method, the navigation to and from a page is noticeable. For example, if the recursive back needs to navigate back through several pages, each of which has its own UI and app bar, the OnNavigatedTo and OnNavigatedFrom methods are executed for each page, which takes time. This time accumulates across the pages and creates a noticeable flick as the automatic recursive back occurs.

To avoid the noticeable flicks, the NonLinearNavigationService sets the opacity of the RootFrame to zero, rendering the RootFrame element and all its children transparent. Therefore, during the recursive back navigation, the application shows nothing.

In addition, **NonLinearNavigationService** exposes the **IsRecursiveBackNavigation** property that is set to true during a recursive back navigation. You can test this property on each page **OnNavigatedTo** and **OnNavigatedFrom** methods to avoid any unnecessary work that will delay the recursive back. We don’t want unnecessary work to get executed on a page that we are “just passing through” on our way to the beginning of the loop. Therefore you should use the following code snippet in your methods:

if (NonLinearNavigationService.Instance.IsRecursiveBackNavigation == true)

{

return;

}

Employing these two techniques yields a very effective solution for recursive back navigation and for solving navigation loops.

## NonLinearNavigationService Limitations

There are few limitations that you need to be aware of when using the NonLinearNavigationService:

* NLNS will not allow you to create any navigation loops whatsoever; even if you want to create a loop, there is no way to signal to the **NonLinearNavigationService** that you want it
* If you are using an application bar in your application, you will have to hide it manually in the **OnNavigatedFrom** method to avoid flickering during the recursive back; while the **NonLinearNavigationService** sets the **RootFrame** to transparent during the recursive back navigation, the app bar requires special attention
* A bug that was initiated by implementation of the Back button manifests when canceling navigation requests; to work around this, you need to use the following code snippet in all of your pages if you are manually canceling the navigation event (usually in the **OnNavigatedFrom** method):

protected override void OnBackKeyPress(System.ComponentModel.CancelEventArgs e)

{

base.OnBackKeyPress(e);

// due to bug, BackKey doesnt send navigation events so we handle this myself

if (NavigationService.CanGoBack)

{

e.Cancel = true;

NavigationService.GoBack();

}

}

## Behind the Scenes of the NonLinearNavigationService

This section explains the internal implementation of the **NonLinearNavigationService**.

The **NonLinearNavigationService** is a singleton with a lazy static constructor. Having multiple instances of the **NonLinearNavigationService** will most likely result in errors, as multiple navigation requests will be initiated at the same time, which is something the phone system doesn’t support.

The **NonLinearNavigationService** keeps a reference, \_**AppRootFrame**, to the application **RootFrame** element to enable easy access to the application navigation service and to set the **RootFrame** transparency.

The **NonLinearNavigationService** has a history list, \_**History**, of type **List<Uri>** that holds the list of pages viewed by the user. This is essentially a copy of the application’s page back-stack.

The **NonLinearNavigationService** has a public property, **IsRecursiveBackNavigation**, that is set to true during recursive back navigation and is used to signal that the application is performing a recursive back navigation.

To Initialize the service, you need to call the **Initialize()** method. This method expects a **PhoneApplicationFrame** object as a parameter. The application constructor is the best location within the application to set the service. Use the following code snippet:

// init the non linear nav service, make sure you call it last in the app CTOR

NonLinearNavigationService.Instance.Initialize(RootFrame);

Looking at the implementation of the Initialize () method, you’ll see that we are setting an event handler to the **RootFrame** **NavigatedTo** method. This method is executed just before navigation to the application’s first page during the application launch process. This ensures that the **RootFrame** will be valid and that we’ll be able to set an event handler to it. This event handler is where most of the **NonNavigationServices** initialization code executes, as shown in the following code snippet:

private void root\_Navigated(object sender, NavigationEventArgs e)

{

Debug.WriteLine("in NonLinear root\_Navigated = " + \_AppRootFrame.Source.ToString());

\_AppRootFrame.Navigated -= root\_Navigated;

\_History = LoadServiceHistory();

// set the current to the first page on which the navigation is used and created...

// unless returning from tombstone where the uri already exists...

if (!\_History.Contains(e.Uri))

{

\_History.Add(\_AppRootFrame.Source);

}

// bind to application's navigation event

\_AppRootFrame.Navigated += new NavigatedEventHandler(NonLinearNavigationService\_Navigated);

\_AppRootFrame.Navigating += new NavigatingCancelEventHandler(NonLinearNavigationService\_Navigating);

\_AppRootFrame.FragmentNavigation += new FragmentNavigationEventHandler(NonLinearNavigationService\_FragmentNavigation);

\_AppRootFrame.NavigationFailed += new NavigationFailedEventHandler(NonLinearNavigationService\_NavigationFailed);

\_AppRootFrame.NavigationStopped += new NavigationStoppedEventHandler(NonLinearNavigationService\_NavigationStopped);

// set our history reference to the global property bag - state for tombstoning

PhoneApplicationService.Current.State[\_HistoryStateKey] = \_History;

}

In this code you can see that first we removed the event handler from the RootFrame Navigated event to avoid registering twice for the same navigation events. Next we loaded the navigation history using the LoadServiceHistory helper method. This method makes sure we get the most up to date history. Also notice that the last line of this method placed a reference to the history list, \_History, in the application State dictionary. This is done to properly handle application tombstoning.

You may also notice that we attached an event handler to each navigation event that the phone **NavigationServices** exposes. In these handlers you will find the logic of the NonLinearNavigationService, and you are more than welcome to review them.

# Summary

The UX recommendation is to avoid navigation loops, and avoid features like having a home button. However, sometime there is no choice but to have these navigation loops. For such scenarios you should use the Non-Linear Navigation Service for Windows Phone.