**HOW DO I …**

Use FAS and Tombstoning in my application?

This is the second of two videos. In the previous video, we saw how to turn an XNA game into a Silverlight/XNA hybrid application. In this video we will see how to add tombstoning and FAS support to this new application.

# Review

In case it has been a while since you saw the first video, or if you haven’t seen it at all, we’ll start by reviewing what we did last time.

We started off with a fully functional game which we upgraded to use Windows Phone Mango in order to demonstrate the benefits of fast application switching. However, we noticed that returning to the game is too sudden and decided to turn the game into a hybrid application. The reasoning behind the transition was that using Silverlight’s page navigation, available in hybrid applications, makes adding screens much simpler.

# Adding a pause screen

Currently we only have two screens, the main menu and gameplay screens with each screen reachable from the other.

We will now add a pause screen, which will allow the user to pause the game and return to the main menu. The pause screen will also be the first screen the user sees when returning to the game after it has been deactivated, allowing the user to prepare before resuming the game.

1. Start by creating a copy of the main menu page. Rename the new page to “PausePage” and make the necessary renaming operations inside the xaml and xaml.cs files.
2. Replace the “ContentPanel” Grid element on the page with the following:

<Grid x:Name="ContentPanel" VerticalAlignment="Center" Grid.Row="1"

Margin="12,0,12,0">

    <Grid.RowDefinitions>

        <RowDefinition/>

        <RowDefinition/>

    </Grid.RowDefinitions>

    <Button Grid.Row="0" Height="100" Content="RESUME GAME"

Click="Resume\_Click" />

    <Button Grid.Row="1" Height="100" Content="MAIN MENU"

Click="Menu\_Click" />

</Grid>

1. Open the code file for the pause page, **PausePage.xaml.cs**, and replace the **Button\_Click** handler with the following methods:

// Simple button Click event handler to take us to the game page

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

{

    NavigationService.GoBack();

}

// Simple button Click event handler to take us to the main menu

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

{

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

}

1. Next, open **GameplayHelper.cs** and navigate to the **HandleInput** method. Locate the comment which reads “// There is no pause screen yet” and replace it and the line beneath it with:

return false;

1. Open the **GamePage.xaml.cs** file and navigate to the **HandleInput** method. Add the following code after the only “if” clause in the method:

else if (shouldPause)

{

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

}

1. On the same file, navigate to the **OnNavigatedTo** handler and replace the call to **InitializeGame** with the following:

if (e.NavigationMode != NavigationMode.Back)

{

    InitializeGame();

}

1. Navigate to the **OnNavigatedFrom** handler and remove the call to gameplayHelper.UnloadContent. We will add it back later on.
2. Finally, navigate to the **MainPage.xaml.cs** file and override the **OnNavigatedTo** handler thusly:

protected override void OnNavigatedTo(

System.Windows.Navigation.NavigationEventArgs e)

{

    // Backing out of this page should always exit the game

    while (NavigationService.CanGoBack)

    {

        NavigationService.RemoveBackEntry();

    }

    base.OnNavigatedTo(e);

}

1. Run the game and the pause screen should now be functional.

# Adding tombstoning & FAS

Now that we have a functional pause screen, it’s time to add support for proper tombstoning and FAS. We will take advantage of the fact that the GameplayHelper class is serializable and that it actually represents the game’s current state. Will preserve the game’s state upon deactivation by using the State dictionary available through the phone application services. Items placed in the State dictionary are persisted between deactivations and activations of the game. Additionally, we will use IsolatedStorageSettings.ApplicationSettings, which is roughly equivalent to the State dictionary but is backed up by isolated storage, to be sure that even if our game is no longer available for activation after being deactivated, the user will still be able to resume the game.

1. Start by modifying **App.xaml.cs** and add the following definitions to the **App** class:

public const string GameStateKey = "GameState";

public bool JumpToPauseScreen { get; set; }

/// <summary>

/// Whether or not asset reloading is required.

/// </summary>

public bool ReloadRequired { get; set; }

1. Add a using directive for “System.IO.IsolatedStorage”.
2. Place the following code inside the **Application\_Launching** method:

if (IsolatedStorageSettings.ApplicationSettings.Contains(App.GameStateKey))

{

    // Restore the application's state from isolated storage

    PhoneApplicationService.Current.State[App.GameStateKey] =

        IsolatedStorageSettings.ApplicationSettings[App.GameStateKey];

    JumpToPauseScreen = true;

ReloadRequired = true;

}

1. Place the following code inside the **Application\_Activated** method:

JumpToPauseScreen = true;

ReloadRequired = !e.IsApplicationInstancePreserved;

1. Place the following code inside the  **Application\_Deactivated** method:

if (PhoneApplicationService.Current.State.ContainsKey(App.GameStateKey))

{

    // To be on the safe side, store the application's state in

// isolated storage

    IsolatedStorageSettings.ApplicationSettings[App.GameStateKey] =

        PhoneApplicationService.Current.State[App.GameStateKey];

}

1. Place the following code inside the **Application\_Closing** method:

// When we exit the application intentionally, there is no point in

// keeping state data

IsolatedStorageSettings.ApplicationSettings.Clear();

1. Next we’ll alter the pause screen so that once we reach it after restoring the game we mark that automatic navigation successful. Open **PausePage.xaml.cs** and add the following **OnNavigatedTo** override:

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

{

    (App.Current as App).JumpToPauseScreen = false;

    base.OnNavigatedTo(e);

}

1. Open the **MainPage.xaml.cs** file and add a using directive for **Microsoft.Phone.Shell**.
2. Add code for cleaning the preserved game state when returning to the main menu, as well as jumping to the pause screen when necessary. Add this code at the beginning of the **OnNavigatedTo** override:

if ((App.Current as App).JumpToPauseScreen)

{

    // The game page, in turn, will jump to the pause page

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

    base.OnNavigatedTo(e);

    return;

}

if (PhoneApplicationService.Current.State.ContainsKey(App.GameStateKey))

{

    (PhoneApplicationService.Current.State[App.GameStateKey] as

GameplayHelper).UnloadContent();

    PhoneApplicationService.Current.State.Remove(App.GameStateKey);

}

1. All that is left is to fix the gameplay screen. Open **GamePage.xaml.cs** and add a using directive for **Microsoft.Phone.Shell**.
2. Modify the **OnNavigatedTo** override with code that handles jumping to the pause screen if necessary. Add the code at the top of the override:

if ((App.Current as App).JumpToPauseScreen)

{

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

    base.OnNavigatedTo(e);

    return;

}

1. Modify the override further by replacing the call to **InitializeGame** and the condition surrounding it with the following code:

if (!PhoneApplicationService.Current.State.ContainsKey(App.GameStateKey))

{

    InitializeGame();

}

else

{

    ResumeGame();

}

1. Add the new **ResumeGame** method used in the above snippet:

private void ResumeGame()

{

    if ((App.Current as App).ReloadRequired)

    {

        gameplayHelper =

PhoneApplicationService.Current.State[App.GameStateKey] as

GameplayHelper;

        gameplayHelper.InitializeAssets(contentManager, spriteBatch,

            SharedGraphicsDeviceManager.Current.GraphicsDevice);

        gameplayHelper.LoadContent();

    }

}

1. Lastly, add the following code inside the **OnNavigatedFrom** override, just before the call to the base implementation:

// When navigating away from the page, save the gameplay state

// (unless we haven't restored it yet)

if (!(App.Current as App).JumpToPauseScreen)

{

    PhoneApplicationService.Current.State[App.GameStateKey] = gameplayHelper;

}

# Summary

In this last of two videos, we’ve seen how to take a previously created Silverlight/XNA hybrid application and use the navigation model exposed by Silverlight to easily add an additional screen so that we may properly handle tombstoning and fast application switching. The techniques demonstrated in this video are not exclusive to Silverlight/XNA hybrid applications and can be used in any of your applications.