Best practice tips for delivering apps to Windows Phone with 256 MB

This article provides recommended best practices for preparing your Windows Phone apps for the Nokia Lumia 610 and other 256 MB Windows Phone devices.

04 Mar 2012

Introduction

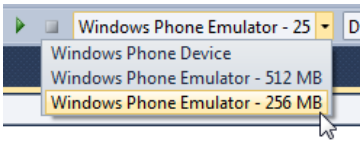
To enable apps to run on 256 MB Windows Phone products a number of changes have been made.

Firstly, the way that memory is used/allocated is different on 256 MB phones when compared to 512 MB devices. Apps running on 256 MB phones will still have the same total amount of memory available (90 MB), but everything after the first 60 MB "Working set" will be paged. Therefore while it is permissible for apps to use up to 90 MB, apps that use less than 60 MB may perform better. Secondly, support for scheduled tasks with potentially unbounded memory consumption are not supported on 256 MB devices.

This article provide some best practice tips and techniques that can be used to achieve the 60 MB goal and to work with the other minor platform changes.

Tip #1 - Always test apps using the emulator's 256 MB option

The emulator supplied with the Windows Phone SDK 7.1.1 gives you the option to choose between emulation of 256 MB and 512 MB memory. Once you have selected a memory option the emulator exhibits the same memory allocation behaviour as physical phones.



The recommended best practice is to always use the 256 MB emulator as your default emulator for all application development. Taking this approach ensures that any issues with your app working well on 256 MB will be dealt with early in your development. In addition, while the emulator is good enough to simulate real memory allocation scenarios, we also recommend you test on a real 256MB device if at all possible.

Tip #2 - Make use of the Windows Phone Memory profiler

The Windows Phone SDK 7.1 includes the Windows Phone Memory Profiler. This tool enables you to see a graph of the current memory allocations, analyze memory use over specific time periods, sees actionable recommendations, and view a dump of the managed heap. The Memory Profiler is available on all Visual Studio versions.

Check out the Techniques for memory analysis of Windows Phone apps article for more information on finding out about your app's memory use.

Tip #3 - Create a helper class to detect if your app is running on a 256 MB phone

Windows Phone 7.5 now includes a property to retrieve the maximum working set available to an app on the phone it's running on. If that maximum is less than 90 MB (or more specifically 94371840 bytes) then the app should consider it to be a 256 MB phone. We recommended you create an all-purpose property that helps you write `if-then-else` conditions for low memory phones, as follows:

```
public static class LowMemoryHelper
{
    public static bool IsLowMemDevice { get; set; }

    static LowMemoryHelper()
    {
        try
        {
            Int64 result = (Int64)DeviceExtendedProperties.GetValue("ApplicationWorkingSetLimit");
            if (result < 94371840L)
                IsLowMemDevice = true;
            else
                IsLowMemDevice = false;
        }
        catch (ArgumentOutOfRangeException)
        {
            // Windows Phone OS update not installed, which indicates a 512-MB device.
            IsLowMemDevice = false;
        }
    }
}
```

For example we could write the following code:

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
```

 (C) Copyright Nokia 2013. All rights reserved.

```
    if (!LowMemoryHelper.IsLowMemDevice)
        Allocate80MbOfMemory();
    else
        DontAllocate80MbOfMemory();
}
```

The goal is not to use `if-then-elses` in your code for 256 MB phones, but sometimes its unavoidable.

Tip #4 - PeriodicTask and ResourceIntensiveTasks are not supported

Both the `PeriodicTask` and `ResourceIntensiveTask` classes aren't supported on 256 MB phones and trying to schedule them will `throw` an exception. These two classes are used to execute developer written code as a background process within certain limitations. It's easy to see why these classes aren't supported. The `ResourceIntensiveTask` can run any code with no upper memory limit. Now imagine you have a 256 MB device with that's using approximately 100MB for the OS, an app with up to 60 MB working set and another background task that's running another 60 MB working set. That's dangerously close to crashing the phone. Activating another background task (for example, 15 MB for background audio) will cause out of memory conditions for the whole phone.

$$\underset{\text{(Operating system)}}{100\text{MB}} + \underset{\text{(Foreground app)}}{60\text{MB}} + \underset{\text{(ResourceIntensiveTask)}}{60\text{MB}} + \underset{\text{(AudioStreamingAgent)}}{15\text{MB}} \approx 256\text{MB}$$

A similar calculation explains why `PeriodicTask` isn't available. With 10 background agents at 6MB each, having 10 `PeriodicTasks` is equivalent to having 60 MB of memory used by another app. For more information on `PeriodicTask` and `ResourceIntensiveTask` see MSDN's Background Agents Overview for Windows Phone. It's important to note that `BackgroundTasks` for background audio and background file transfer as well as Scheduled Alarms and Reminders will continue to work on 256 MB phones.

Tip: The app can still contain code for `PeriodicTask` and `ResourceIntensiveTask`. The exception is thrown when you attempt to schedule them.

The recommended best practice is to always `if-then-else` your code using the code in best practice #3 and not initialise `PeriodicTask` or `ResourceIntensiveTasks` on 256 MB phones, as follows:

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    if (!LowMemoryHelper.IsLowMemDevice)
        InitializePeriodicTaskToUpdateLiveTiles();
    else
        InitializePushNotificationsToUpdateLiveTiles();
}
```

For some use-cases it will be possible to use other means to compensate for the missing featureset. For example `PeriodicTask` powered Live Tiles can be replaced with Push Notification powered Live Tiles.

In Windows Phone 7.5 Background Agents are only meant to provide additional app functionality and not core app functionality as users might turn them off. However, if your app is centered around a `PeriodicTask` or a `ResourceIntensiveTask` it's best for it to be excluded from supporting 256 MB apps.

Tip #5 - Use the WebBrowserTask instead of the <WebBrowser /> control to display arbitrary untested web pages

In Windows Phone 7 it's possible to load up any URL into your app using the Internet Explorer `<WebBrowser />` control. However, some web-pages could cause excessive memory consumption. Websites that aren't tailored to mobile web browsers in particular might cause significant memory footprints on mobile phones.

For example, a `<WebBrowser />` pointing to a webpage containing code that hasn't been customized for mobile browsers, such as

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <phone:WebBrowser Source="http://www.yvettesbridalformal.com"
                          VerticalAlignment="Stretch"
                          HorizontalAlignment="Stretch" />
</Grid>
```

Will cause a spike in memory usage.

Note that not all websites will be cause issues on 256 MB phones. For example, memory consumption will still be within acceptable limits for a more modern website, as shown below:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <phone:WebBrowser Source="http://developer.nokia.com"
                          VerticalAlignment="Stretch"
                          HorizontalAlignment="Stretch" />
</Grid>
```

Even when navigating to tested websites, it's important to make sure they only navigate to previously tested pages. One option is to block navigation to websites known to use an excessive amount of memory. As a general rule it's better to limit navigation to known websites that have been tested successfully, and not limit against specific websites.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <phone:WebBrowser Source="http://developer.nokia.com"
                          Navigating="WebBrowser_Navigating"
                          VerticalAlignment="Stretch"
                          HorizontalAlignment="Stretch" />
```

 (C) Copyright Nokia 2013. All rights reserved.

```
</Grid>
```

```
private void WebBrowser_Navigating(object sender, NavigatingEventArgs e)
{
    if (e.Uri.OriginalString == "http://www.yvettesbridalformal.com")
    {
        e.Cancel = true;
    }
}
```

To guarantee your app won't have memory issues on 256 MB phones due to the `<WebBrowser />` control you can simply always use the `WebBrowserTask`. The `WebBrowserTask` will open up a separate app and will tombstone your app if memory gets tight. So make sure to build tombstoning support into your app.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    new WebBrowserTask()
        {
            Uri = new Uri("http://developer.nokia.com", UriKind.Absolute)
        }.Show();
}
```

The recommended best practice for the `<WebBrowser />` control is to test all pages that could potentially run inside a `<WebBrowser />` in your app under memory profiler. Make sure that at no point memory consumption is over 90 MB. If you cannot limit external links from your app (as would be the case in a Reddit style app) don't use the `<WebBrowser />` control, instead use the WebBrowserTask. Another possibility is a hybrid approach where you monitor `<WebBrowser />` memory usage, and if it goes above 90 MB then you remove the control from the visual tree and launch a `WebBrowserTask`.

Tip #6 - Replace the Bing <Map /> control with the BingMapsTask

The Bing `<Map />` control loads maps. Maps are composed of many small bitmap images that vary depending on the map's latitude & longitude location, your zoom factor, the map type, and a few other factors. That's quite a lot of potential bitmaps. Every time a user interacts with a Bing `<Map />` control new bitmaps will likely get downloaded from the Bing Maps server and loaded into memory. Even a simple Maps control could likely load many images.

```
<m:Map
    xmlns:m="clr-namespace:Microsoft.Phone.Controls.Maps;assembly=Microsoft.Phone.Controls.Maps"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"/>
```

After navigating with this map for two minutes you will see memory is almost at 50 MB, many images were initialised and 5 different garbage collections took place. 50 MB is a lot less of an issue on a 90 MB working set for 512 MB phones than it is for the 60 MB working set on 256 MB devices. So Bing `<Map />` control requires more of your attention on 256 MB phones.

One possible quick and dirty fix is to make the Bing maps control non-interactive.

```
<m:Map
    xmlns:m="clr-namespace:Microsoft.Phone.Controls.Maps;assembly=Microsoft.Phone.Controls.Maps"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"
    IsHitTestVisible="False" />
```

By setting `IsHitTestVisible=False` you're essentially saying that the Bing Maps control is static. By disallowing user navigation in the Bing `<Maps />` control you're essentially stopping the excessive image loading this control might perform. The only images that will get loaded are for those initial properties set by you.

However, if you need to allow the user to navigate in the map, that quick and dirty won't work. For that use-case it's recommended you use the `BingMapsTask`. The `BingMapsTask` will open up a separate app and tombstone your app when memory gets tight. So make sure to build tombstoning support into your app.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    new BingMapsTask()
        {
            SearchTerm = "Espoo, Finland"
        }.Show();
}
```

The recommended best practice for the Bing `<Map />` control to replace it for the `BingMapsTask`. If that's not possible or a simpler static solution is preferred set `{{{1}}}` on the Bing `<Map />` control. Another possibility is a hybrid approach where you monitor the Bing `<Map/>` control memory usage, and if it goes above 90 MB then you remove the control from the visual tree and launch a `BingMapsTask`.

Tip #7 - Consider reducing image quality

Images will consume as least as much size on memory as their size on disk. Any excessive use of non-mobile optimized images will inevitably cause a spike in memory use. There are many things that could be done to reduce image memory footprint: Use a 480x800 maximum image size, choose a format wisely (PNG or JPG) and reduce quality where needed.

For example, the following URL has a 4913x3400 image @ http://tinyurl.com/nokia8mp

 (C) Copyright Nokia 2013. All rights reserved.

You can see the top-right `<MemoryCounter />` shows 13MB-16MB of memory with this image loaded, instead of 6MB of memory for a default Windows Phone 7 app. That's 8MB of memory that was allocated. You could potentially scale down the image to the maximum resolution on Windows Phone 7.5 and save a lot of that memory.

Let's scale down the image to a width of 800 pixels which is the absolute maximum width any image we'll need on WP7. By limiting the image to this maximum resolution we shouldn't lose any quality, have an identical user-experience but a lower memory footprint.

We can see that just by resizing this image from 4913x3400 to 800x554 we've brought the total memory footprint down from 13-17MB to 9-13MB. The recommended best practice for `<Image />` controls is therefore to opt-in for lower resolution images whenever possible. If a lower resolution image isn't available then depending on your circumstances it might be best to not show the image at all or to attempt to resize it on the server.

Tip #8 - Consider replacing long lists with images with ListBox replacements that use Data Virtualization

 (C) Copyright Nokia 2013. All rights reserved.

As we've just seen images could potentially take some memory. But since one image takes a noticeable amount of memory, then a long list with images must take considerably more memory. Let's see an example of that with a long list of Flickr images.

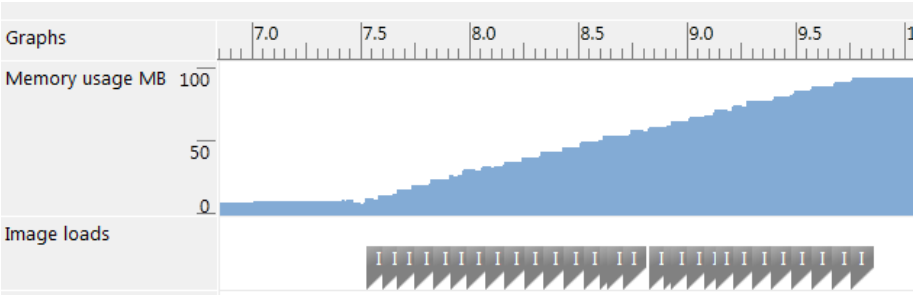
As our first step we'll download the package for Flickr.net API and add a reference to the `FlickrNetWP7` assembly. Next we'll search for Flickr images and add those to the UI:

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    Flickr flickr = new Flickr("<flickr API token>", "<flickr API secret>");
    flickr.PhotosSearchAsync(new PhotoSearchOptions(null, "nokia"),
        result =>
        {
            Dispatcher.BeginInvoke(() =>
                lst.ItemsSource = result.Result);
        });
}
```

We'll display that list of images with their respective title:

```
<ListBox x:Name="lst" VerticalAlignment="Stretch" HorizontalAlignment="Stretch">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Image Source="{Binding LargeUrl}" Width="200" Stretch="UniformToFill"/>
                <TextBlock Text="{Binding Title}" Margin="2" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Running this app and memory profiling it you could see that memory goes well over 90 MB of memory. You could even see the little "steps" on memory usage with each new image initialized.



One approach here would be to limit the number of images shown to the top 10 images and then page for a separate page. Another option would be move from using large Flickr images to smaller Flickr images. In our example we'll limit the app to only show 10 smaller Flickr images at the same time.

```
<ListBox x:Name="lst" VerticalAlignment="Stretch" HorizontalAlignment="Stretch">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Image Source="{Binding SmallUrl}" Width="200" Stretch="UniformToFill"/>
                <TextBlock Text="{Binding Title}" Margin="2" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    Flickr flickr = new Flickr("2191ef82aa075c112349ff21c45f4b27", "224975f561e65fc4");
    flickr.PhotosSearchAsync(new PhotoSearchOptions(null, "amazing everyday"),
        result =>
        {
            Dispatcher.BeginInvoke(() =>
                lst.ItemsSource = result.Result.Take(10));
        });
}
```

When we run the memory profiler for this app we can see a much more reasonable memory consumption.

We've changed the memory footprint of the app with very limited functionality changes. Off course for each app you'll need to make whatever adjustments for long image lists as make sense. The recommended best practice is for app developers to review any long list with images, memory profile those pages and make UX changes where required. A strong first step would be to memory profile ListBox replacements that use Data Virtualization, such as Peter Torr's LazyListBox or David Anson's DeferredLoadListBox. If just changing the ListBox control variant doesn't work then you'll need to consider UX changes. For example consider using Paging, placing different categories of data into different pages or using a Silverlight Toolkit LongListSelector. UX changes depend on your specific business domain and UX requirements.

Tip #9 - Consider disabling page transitions

Page transitions are the animations that take place when the user navigates between pages (such as flip-in and flip-out). On WP7 those animations are commonly enabled through the Silverlight Toolkit for Windows Phone using the `TransitionFrame` and `TransitionService`. In this section we'll focus on the memory impacts of using the `TransitionFrame` and `TransitionService`. We'll see that memory consumption of page transitions is about 5MB~. Those 5MB~ could turn out to be important when you have 60 MB working set versus 90 MB working set. We should call out that any and all implementations of Page Transitions would cause this memory footprint, and not just the Silverlight Toolkit implementation. Any implementation of having 2 pages animate at

 (C) Copyright Nokia 2013. All rights reserved.

the same time will inherently require considerable memory.

We'll start off our example by downloading the Silverlight Toolkit for Windows Phone. You can either setup the Silverlight Toolkit using NuGet, or install the MSI and add a reference to the `Microsoft.Phone.Controls.Toolkit` assembly. Once you've done that you'll need to make two changes to your app. The first is in App.xaml.cs where instead of using the plain old `PhoneApplicationFrame` we should use the `TransitionFrame`.

```
//RootFrame = new PhoneApplicationFrame();
RootFrame = new TransitionFrame();
```

For each page we'd like to enable page transitions for we'll add the following XAML specifying which animations should happen where. For more information on how to use the `TransitionFrame` see WindowsPhoneGeek's Windows Phone 7 Navigation Transitions Step By Step guide.

```
<toolkit:TransitionService.NavigationInTransition>
    <toolkit:NavigationInTransition>
        <toolkit:NavigationInTransition.Backward>
            <toolkit:TurnstileTransition Mode="BackwardIn"/>
        </toolkit:NavigationInTransition.Backward>
        <toolkit:NavigationInTransition.Forward>
            <toolkit:TurnstileTransition Mode="ForwardIn"/>
        </toolkit:NavigationInTransition.Forward>
    </toolkit:NavigationInTransition>
</toolkit:TransitionService.NavigationInTransition>
<toolkit:TransitionService.NavigationOutTransition>
    <toolkit:NavigationOutTransition>
        <toolkit:NavigationOutTransition.Backward>
            <toolkit:TurnstileTransition Mode="BackwardOut"/>
        </toolkit:NavigationOutTransition.Backward>
        <toolkit:NavigationOutTransition.Forward>
            <toolkit:TurnstileTransition Mode="ForwardOut"/>
        </toolkit:NavigationOutTransition.Forward>
    </toolkit:NavigationOutTransition>
</toolkit:TransitionService.NavigationOutTransition>
```

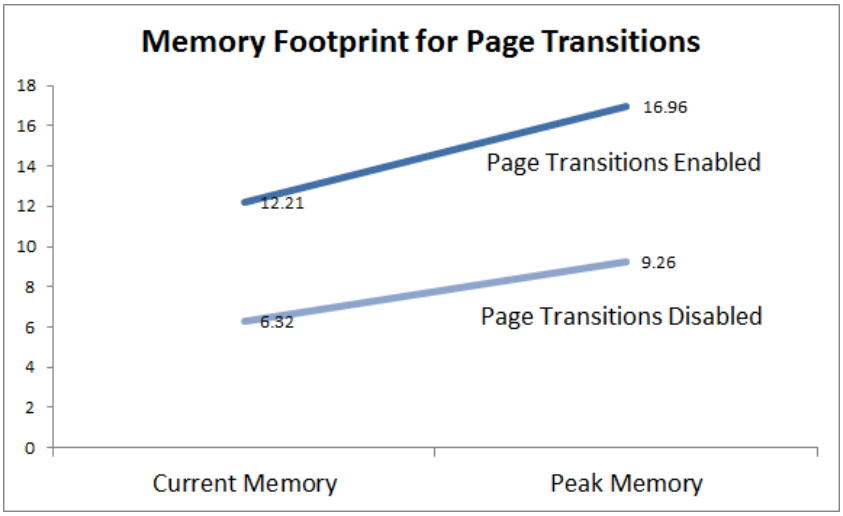
When running this application we see that current memory is at ~12 MB and peak memory is at ~17MB.

If we disable page transitions we can see a modest reduction in current memory, but a slightly more significant reduction in peak memory usage. We can disable Page Transitions by using the original `PageTransitionFrame` and not use the `TransitionFrame`.

```
RootFrame = new PhoneApplicationFrame();
//RootFrame = new TransitionFrame();
```

When we run the same app now with page transitions disabled we can see that the memory footprint is somewhat reduced.

We can see the results of our simplified ad-hoc experiment much more clearly on this graph:



While Page Transitions are great to enable the Metro UI principle of "fast and fluid", it's not so great to lose 5MB-7MB of memory when we've only got 60 MB of memory to work with. It's a delicate balancing act. For most apps it will make sense to disable page transitions completely for 256 MB devices unless you can guarantee overall app memory usage isn't over 55MB~.

```
if (LowMemoryHelper.IsLowMemDevice)
{
    RootFrame = new PhoneApplicationFrame();
}
else
{
    RootFrame = new TransitionFrame();
}
```

The recommended best practice is for app developers to disable page transitions on 256 MB devices unless they've tested their apps on 256 MB devices and overall memory usage doesn't exceed 90 MB.

Tip #10 - Avoid initialising the same SoundEffects multiple times

Many XNA games and even a few Silverlight apps use the `SoundEffect` class to play short audio clips. The most common use case is for sound effects in games. Such effects could be sound effects for shots fired, movement, collisions and hits. Those are events that happen very frequently in games.

 (C) Copyright Nokia 2013. All rights reserved.

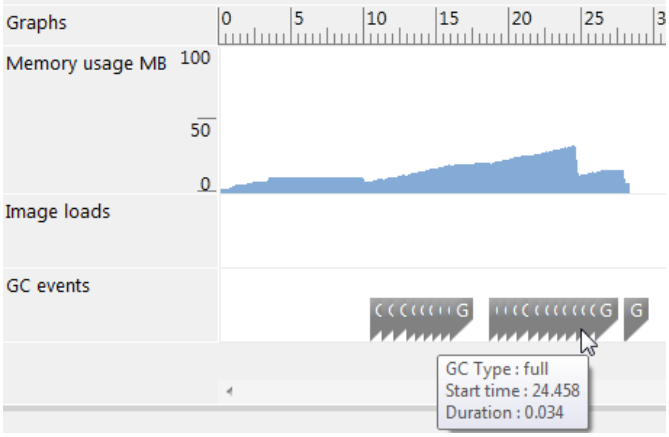
However, initializing multiple SoundEffects and not disposing of them would cause a considerable temporary memory hit. For more information on SoundEffect usage in WP7 see Maarten Struys's Adding Sound Effects to a Windows Phone 7 Silverlight Application.

A worst practice for 256 MB would be to initialize a new SoundEffect every time you need to play a sound.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    SoundEffect beep = SoundEffect.FromStream(
        Application.GetResourceStream(
            new Uri("NokiaBeep.wav", UriKind.RelativeOrAbsolute))
                .Stream);

    FrameworkDispatcher.Update();
    beep.Play();
}
```

If we run this app and click our button many times in short and rapid succession we will see the following memory footprint:



You can see that even though SoundEffects aren't being reused or disposed, the Garbage Collector whenever it does a full GC run will dispose of the SoundEfffect. This situation might be acceptable for 512 MB devices. But this is especially bad for 256 MB devices since this gradual accumulation of memory could easily pass 60 MB before getting garbage collected. In our simplified example we had almost no visuals and no actual game logic we've approached 40MB~ of memory usage. Not disposing or reusing SoundEffects becomes much more noticeable in terms of overall memory footprint when your working set is just 60 MB.

There are many best practices to manage your SoundEffects explored by the XNA community. You can use timers to dispose of soundeffects en masse, you can cache a dictionary of reusable SoundEffects, et cetera. For our simplified example we'll cache the SoundEffect and dispose of it when it is no longer needed when the page is no longer visible.

```
private SoundEffect beep = null;
private void Button_Click(object sender, RoutedEventArgs e)
{
    if (beep == null)
    {
        beep = SoundEffect.FromStream(
        Application.GetResourceStream(
            new Uri("NokiaBeep.wav", UriKind.RelativeOrAbsolute))
                .Stream);
    }

    FrameworkDispatcher.Update();
    beep.Play();
}

protected override void OnNavigatedFrom(System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedFrom(e);
    beep.Dispose();
}
```

By making this small change we're only allocating a single SoundEffect, the app behaviour is identical and we've improved the memory footprint significantly. Instead of allocating many SoundEffects in memory we've only allocated one. The recommended best practice is to avoid initializing the same SoundEffects multiple times, cache initialized SoundEffects while they're needed and dispose of them when they're not.

Tip #11 - Compress your XNA assets

XNA games can load so many assets into memory that so it tips over 90 MB. It might be possible to reduce the overall memory footprint of an item by compressing the vector data or textures used. In the most optimistic scenario you can reduce the assets' size on disk and size in memory while retaining perfect quality and not impacting CPU and GPU performance. In most scenarios however it's likely that data compression will cause a loss of quality in the XNA assets.

There are quite a few options for asset compression depending on what assets your XNA app has. For example, it's possible to save 25% on models (8 bytes on every 32 bytes) by giving up some accuracy in positioning by normalizing vector data. Another possibility is to reduce overall asset quality using DXT compression algorithms. The idea here isn't to "compress" the content's size in memory without trading away considerable CPU or GPU time. This isn't compression meant to reduce size on-disk, but rather size in memory. JPG compression for example wouldn't work for us since it just compresses size on disk; while DXT compression would compress size on disk and size in memory.

All of these compression algorithms will run on the development machine as a pre-packaging step. Additionally the compression algorithms might have to be used by the app when it runs depending on the algorithm. For an introduction to XNA asset compression see Shawn Hargreaves's Compressed GPU

 (C) Copyright Nokia 2013. All rights reserved.

data formats.

One last pointer for XNA games it to plan garbage collection calls (GC.Collect) based on your game loop and current memory consumption based of your application. The recommended best practice is for XNA developers to profile their memory usage and if needed (over 90 MB) explore asset compression strategies. For more information on XNA memory optimization see MSDN's Improving Memory Use in XNA Games.

Tip #12 - Take care to locate and eliminate memory leaks

Memory leaks are caused when normal use of your app memory is unexpectedly and gradually allocating memory, but not deallocating that memory later on. A common telltale of memory leaks is losing a few MB of memory with each page navigation and that memory is not reclaimed until the app is restarted. Memory leaks will exist on 256 MB devices as much as they do on 512 MB devices and there's no difference there. However, due to the lower memory working set (60 MB over 90 MB) they become much more noticeable. e.g. a WP7 app running in 50 MB would be able to leak 40MB on 512 MB devices, but would only be able to leak 10MB of working set memory on 256 MB device. That's considerably less time the app can be used by consumers before the memory leak becomes pronounceable.

For more on how to diagnose memory leak on WP7 see Windows Phone's blog Memory Profiling for Application Performance.

The recommended best practice is to know how to Find Managed Memory Leaks in WPF and Silverlight applications, profile your app during heavy use and listen to your users regarding overall app performance. If users are reporting "the app is getting 'tired' as time progresses" or "after I use the app for an hour the app always crashes" those are telltale signs of a memory leak. Look at what's on the managed heap during a specific period of time and determine if it actually should be there.

 (C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.