# Loading Time Reduction Strategies for XNA Game Studio Titles on Windows Phone 7

By Scott Selfon, Jeff Petkau, Xin Huang, and Cullen Waters  
Advanced Technology Group

*Published: December 8, 2010  
Updated: December 30, 2010*

## Introduction

Applications on mobile devices typically require low-friction entry and exit, allowing for quick and responsive experiences. Unlike console or PC games, where the user can invest hours in a single game session, use of applications on mobile devices is often on the order of seconds to minutes. Users can become frustrated when too much of these briefer experiences are taken up with loading screens and other non-interactive taxes on their desire to interact with their applications. To support this, the [Windows Phone Application Certification Requirements](http://go.microsoft.com/?linkid=9730558) provides requirements for responsiveness at launch time. This white paper discusses the intent of the requirements, as well as several best practice techniques for reducing loading times.

**Note**   Measurements and recommendations in this white paper are based on launch hardware and software characteristics; future system updates and hardware implementations may exhibit different performance traits.

## The Letter—and Intent—of the Law

The key certification elements relating to loading times are documented in section 5.2.1 of [Windows Phone Application Certification Requirements](http://go.microsoft.com/?linkid=9730558); specifically:

“The application must render the first screen within 5 seconds after launch.”

“Within 20 seconds after launch, the application must be responsive to user input.”

The intent of the certification requirements is to enforce consistency and an easy entry to gameplay. Titles should focus on the intent rather than the letter of the law; for example, a 15-second loading time to the initial menu followed by a 2-minute non-interactive loading time to the title would ostensibly meet, but be outside of the spirit of, the certification requirement.

The loading time requirement does apply to both hub-launched and reactivation scenarios, so titles should strongly consider techniques that support minimized loading times throughout the gameplay experience.

## Hardware and Software Characteristics that Affect Loading Times

Loading times are affected by multiple characteristics that can vary with hardware. We have evaluated multiple pre-production and production devices across a series of tests for this white paper. Individual devices may present superior performance, and values here represent practical worst-case values for production hardware used for measurements.

**Note**   Development using pre-production hardware may demonstrate more variable performance due to non-final, performance-limited memory units (as well as pre-chassis 1 specification hardware characteristics)

**Table 1.** Read tests for a 1 megabyte file using various sequential and random access scenarios, as measured over a sample set of fifty or more reads per device.

|  |  |  |
| --- | --- | --- |
| Read Test (1 MB) | 25th percentile performance in MB/sec on slowest device | File shared across tests? |
| Isostore,  random-access 4KB chunks | 0.96 | No |
| Isostore,  random-access 4KB chunks | 1.04 | Yes |
| Isostore,  single block (after random access) | 2.31 | Yes |
| Isostore,  single block (solely sequential reads) | 4.33 | No |
| TitleContainer,  random aligned 4-KB chunks | 1.69 | Yes |
| TitleContainer,  random unaligned 4-KB chunks | 1.27 | Yes |
| TitleContainer,  random sequential 4-KB chunks | 1.14 | Yes |
| TitleContainer,  single block (after random access) | 1.14 | Yes |
| TitleContainer,  single block (solely sequential reads) | 1.15 | No |

Note that random file reads within isostore will cause a file’s later sequential reads to be slower than if the file was strictly sequentially read. Based on launch and preproduction devices seen by ATG, 950 KB/sec presents a reasonable minimum performance number for isostore random access reads, with title storage typically demonstrating 1 MB/sec or better.

## Loading Time Reduction Strategies

### Instrument Your Content Load Sequences

Often the first step in reducing loading times is to understand where the current greatest expenses are. Highlighting the frequency and timing of content loading is an effective way to evaluate and adjust loading times, as well as validate that the right content (no more and no less) is being loaded in a given scenario. Consider instrumenting:

Time required to load an asset

The System.Diagnostics.Stopwatch object can be used for this.

Frequency with which each asset has been loaded

Over multiple game levels, across sessions, and so on.

Frequency with which each asset is freed

Average lifetime.

### Reduce the Size of the Content Data Set: XNA Content Pipeline Compression

Strategies that shrink the amount of content required to be loaded will bring the greatest reductions in loading times. First and foremost, titles should leverage content compression, because this will provide the most significant and immediate reduction of content size. Note that, by default, XNA Game Studio defers to high-fidelity assets (that is, uncompressed), so specifying compression is a task that title developers must explicitly do themselves.

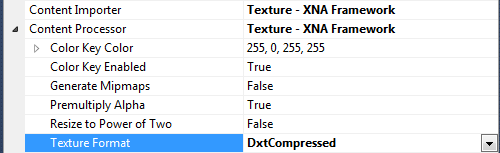
**Figure 1.** Compression settings for a wave file, as seen in the Visual Studio 2010 properties frame. *Best* (the default) means best fidelity, least compression, and largest file size. *Medium* offers better compression, and some assets may acceptably use *low*.



**Table 2.** Audio compression settings for Sound Effect and Music objects.

|  |  |  |
| --- | --- | --- |
| Compression Setting | Compression Applied | Compressed Asset Size, Relative to Uncompressed Source |
| Sound Effect: Best | None | 100% |
| Sound Effect: Medium | ADPCM | approximately 29% |
| Sound Effect: Low | ½ sampling rate, ADPCM | approximately 14% |
| Music: Best | WMA CBR (constant bitrate) 192 kbps | Varies with source format; for 44.1 kHz 16-bit PCM, approximately 27% |
| Music: Medium | WMA CBR 128 kbps | For 44.1 kHz 16-bit PCM, approximately 18% |
| Music: Low | WMA CBR 64 kbps | For 44.1 kHz 16-bit PCM, approximately 9% |

**Figure 2.** Texture format settings, as seen in the Visual Studio 2010 properties frame. *Color* (the default) indicates a 32-bit per pixel RGB image. *DxtCompressed* utilizes DXT1 or DXT5 compression, with substantial space savings both for storage and for runtime memory footprint.



**Table 3.** Texture format settings and their effect on texture sizes.

|  |  |  |
| --- | --- | --- |
| Texture Setting | Processing Applied | Output Asset Size, Relative to Uncompressed Source |
| Texture: NoChange | Preserves existing format for .dds; for PNG or JPEG, returns uncompressed 32-bit per pixel data | Varies with header:data ratio |
| Texture: Color | RGB 32-bit per pixel (raw) | Potentially larger than source |
| Texture: DxtCompressed | DXT1 (if opaque), or DXT5 (if transparent) | DXT1: 12.5% DXT5: 25% |

Of course, the use of compression requires the title developer to make tradeoffs between size and the perceived quality of the assets; some content can be more aggressively compressed without objectionable artifacts, while other content may need preprocessing or to be less aggressively compressed to maintain appropriate fidelity.

### Reduce the Size of the Content Data Set: Formatting, Preprocessing, and Offline Compression/Run-Time Decompression

With titles often sharing assets between platforms, the temptation can be to literally reuse textures, sounds, movies, and so on. If a texture will be consistently scaled down for display on a phone, consider performing that scaling offline, rather than taking the processing penalty—and bandwidth and memory overhead—when loading content.

Developers may also want to exploit other texture types, such as PNG, where doing so wouldn’t contribute to already compressed assets. For sparse textures, PNG on Windows Phone will typically demonstrate superior compression to DXT-compressed content that is brought through the XNA content pipeline. To use other texture types, the source files must be copied to the output directory and not compiled in the content pipeline.

Do note that, while DXT-compressed assets can be used natively by Windows Phone 7 GPUs, many formats—including PNG—need to be expanded at run time to a raw format of 32 bits per pixel. This expansion can lead to increased memory overhead compared to DXT compression.

To balance the run-time memory footprint of DXT with the loading time footprint of more aggressive compression formats, developers may choose to apply custom compression and run-time decompression to DXT content (as built by the XNA pipeline into .xnb files), which can lead to significant reduction of loading times. Developers should balance loading time considerations with CPU requirements to decode their custom-encoded content, as well as with memory requirements to handle and manipulate the decompressed data. Offline custom compression and run-time title-managed decompression of DXT content can offer a good balance of reduced size (and thus, reduced loading time) without large run-time memory costs.

Developers can also pack multiple images into a single texture, as demonstrated by the content processor in the [Sprite Sheet](http://create.msdn.com/en-US/education/catalog/sample/sprite_sheet) sample. Sprite sheets avoid DXT power-of-two restrictions imposed by the XNA content processor, and optimize file loading (replacing many small files with one larger one).

In the realm of sound, if native audio assets from a console title are 48 kHz, consider downsampling them to 44.1 kHz (prior to applying the XNA pipeline’s own compression) for use on the phone. This will realize an immediate 8 percent savings (approximately) on storage and reading bandwidth, as well as mild CPU savings for running at the native sampling rate of the Windows Phone device (44.1 kHz).

### Reduce the Size of the Content Data Set: Data Organization and Ordering

Beyond compression, title developers who need to decrease loading times can focus on data organization that focuses efforts on loading content that is needed to drive to an initial interactive state, rather than preparing all possible loaded data. This is particularly important in avoiding the watchdog timer; a title that loads data for too long prior to drawing to the screen risks being terminated by the system. For more information about the watchdog timer for Windows Phone, see [Don't Get Bit: Avoiding the Windows Phone Watchdog](https://developer.xboxlive.com/en-us/windowsphone/development/education/documents/%5bmobile%5d%20Windows_Phone_Watchdog.docx) on App Hub.

Developers should give similar attention to in-game content loading as well. Remember that returning to gameplay from interruptions (SMS, phone, app purchase, and so on) invalidates all previously loaded content; so, consider a fast path back to gameplay.

### Evaluate Asynchronous Background Loading

Even if the game takes substantial setup time, there are numerous techniques to getting the user into some kind of interactive state sooner. Anything from a simplified arcade-style loading screen to cut-scenes, trivia, “did you know” facts, and other low-CPU-impact techniques can be leveraged to help smooth the set up and transition from loading to gameplay.

Loading to an initial menu state or a cut-scene, and then continuing to load additional assets in the background would seem to be appropriate strategies for masking loading times from the consumer. However, **LoadContent** performs byte copies of each loaded texture asset that uses the XNA Content Pipeline, generating garbage. And **LoadContent**, overall, will trigger garbage collection at each megabyte of loaded data. Depending on the actual interactivity of foreground scenes, the potential CPU cost taken by garbage collection may be acceptable; playback of pre-rendered video cut-scenes takes advantage of purpose-built hardware, so CPU use is typically negligible. Similarly, static or intermittently animated menu systems would likely have more success here than attempting to generate CPU-intensive content rendered in-engine during background loading.

### Anticipate and Reduce Garbage Collection Scenarios

While not the top cost in I/O-bound storage reads from devices, garbage collection can contribute to loading times as well, because garbage collection occurs each time the Content Framework loads a piece of content that pushes the memory use over the next 1-megabyte boundary. This makes evaluation and optimization of the overall size of assets being loaded all the more crucial.

There are a number of situations where titles can *inadvertently* generate garbage, as well, through their own processing; for more information on techniques for reducing or avoiding garbage collection, see [Performance Considerations for Windows Phone 7 Games](http://create.msdn.com/education/catalog/article/performance_wp7_games) on App Hub.

### Consider Custom Serialization

Microsoft’s .NET Framework provides an easy to use method for serializing data to disk, using types present in the System.Xml.Serialization namespace. Simplicity always comes with tradeoffs, however; in this case, the tradeoff is file size. The default serialization schema is verbose. The behavior of the XmlSerializer is trivially easy to change, however, and can result in significant savings in file sizes.

For an example, let’s consider the following class definition.

public class TestClass

{

public int count;

public float size;

public bool enabled;

public string LongNameOfAMinorFieldThatDoesntNeedALongNameInTheFile = "test";

}

This class definition, when serialized with the default XmlSerializer, produces the following XML.

<?xml version="1.0"?>

<TestClass xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<count>0</count>

<size>0</size>

<enabled>false</enabled>

<LongNameOfAMinorFieldThatDoesntNeedALongNameInTheFile>test</LongNameOfAMinorFieldThatDoesntNeedALongNameInTheFile>

</TestClass>

As you can see, the XmlSerializer’s default behavior is to treat each public field or property as an XML element. This generates quite a bit of extra data in the file; this XML file uses 332 bytes on disk to serialize four fields. With a few simple changes, we can get significantly smaller files from the XmlSerializer. Consider the following class declaration.

public class TestClass2

{

[XmlAttribute(AttributeName="c")]

public int count;

[XmlAttribute(AttributeName="s")]

public float size;

[XmlAttribute(AttributeName="e")]

public bool enabled;

[XmlAttribute(AttributeName = "l")]

public string LongNameOfAMinorFieldThatDoesntNeedALongNameInTheFile = "test";

}

With the XmlAttribute added to properties, the XmlSerializer treats the field as attributes rather than elements, and gives the attributes alternative names. The resulting XML is the following.

<?xml version="1.0"?>

<TestClass2 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" c="0" s="0" e="false" l="test" />

The serialized file has significantly less wasted text. The file size also shrank to 167 bytes. That’s a savings of roughly 50 percent, and a more reasonable file size to serialize four fields. Modifying your serialization code to prefer XmlAttributes to XmlElements will often result in similar savings. Even if you don’t perform renaming, as we did in this example, you will generally get close to 50 percent reduction, since every XmlElement has to have a closing tag, while attributes don’t.

Avoid using XmlAttribute for complex types, or for collections of types. The space savings are minimal in these cases, and the resulting file is considerably more difficult to read. For larger amounts of data, consider writing custom binary serialization code. In all cases, ensure you time any new code to confirm any realized performance gains over the default serializer settings.

# Wrap Up

Balancing the need to load data necessary for a compelling experience with actually initiating the experience is a challenge that each title faces. Extensive progress bars and other non-interactive states that do not engage the user can lead to frustration and impatience. Use of some of the techniques illustrated here can aid title developers in helping the user get to the fun of their phone’s applications as quickly as possible. If you have questions, comments, or suggestions for this paper, or have found other techniques that you would like to share, please visit the App Hub forums at <http://forums.create.msdn.com>.