

Best Practices for Game Performance on Xbox 360

By Eron Hennessey

XNA Game Studio Documentation

Abstract

Discusses best practices for achieving good performance on Xbox 360 for XNA Game Studio Framework games.

Introduction

The Xbox 360 is a powerful game system, but there are many things that can degrade your game's performance. This article discusses ways you can improve performance for XNA Game Studio games that target the Xbox 360. It also examines how to make your game better looking and more responsive, and how to provide an enjoyable, polished experience for players.

There are two major ways to improve game performance: improving the performance of your game's *design*, and improving the performance of your game's *code*. In general, a game's design can have the greatest impact on its performance. Once you have a good design, improving the efficiency of the code itself can provide you with additional gains.

Improving Performance in Game Design

By designing your game with performance considerations in mind, you can increase your game's performance even before a single line of code is written. When designing your game, consider the points discussed in the following sections. Both your game's developers and players will thank you!

Focus your attention on the same things your players focus on

Not all game elements are created equal. The areas of your game on which the player's attention is focused will benefit more from high performance and detail than from background elements. Carefully consider what the most important elements of your game are, and present these with the highest level of detail. Other elements in your game can be rendered with less fidelity or can receive fewer update cycles than those at the center of attention.

Just as gamer focus should be considered when designing your game's graphics and sound elements, this consideration should also be applied to game mechanics. In an action game, for instance, silky-smooth action and responsive controls are more important than high-resolution graphics and high-fidelity sound. In a strategy game, spending processing time on enemy AI is more important than spending it on fluid character animation. When displaying a menu, providing a smooth, responsive menu experience matters more than packing the display with special effects and sound.

Update and draw only what is necessary

Design your game to only update and draw elements that the player can see or hear or those that are important for gameplay. Off-screen or obscured elements need not be drawn at all. Any game elements that are not experienced by the gamer may not need to be updated unless they are important to gameplay—for instance, in the case of off-screen enemy units in a strategy game. Even in the case of enemy units, your game might benefit from treating a collection of nearby units as a single entity when they are off-screen, and individually only when they are on-screen. Remove elements that are no longer necessary: enemy corpses can be removed from the battlefield over time. By focusing on the player's attention, you might design your system to remove corpses that cannot be seen because they are off-screen or obscured, or those that are farther from, or older than, those at the center of attention.

Make good use of limited memory

Because many game platforms, including Windows Phone and Xbox 360, have a limited amount of memory available for graphic, sound, and other game data, use this memory to the greatest effect. Think about what game assets need to be loaded into memory, and load them only when they are needed or when they soon will be needed (often referred to as content *preloading*). Game assets such as a font texture used to render many game displays and menus that are used frequently can be kept in memory throughout the game. Other assets, such as location or level-dependent graphics and sounds, can be loaded when the player nears that location or begins the level.

As mentioned earlier, thinking about what elements your player will focus on can also help you determine which assets can be stored at lower resolution and which should be provided more memory to provide your players with the best experience. In a card game, the faces of the cards will receive more attention than the card backs and the card table, both of which can be represented by lower-resolution textures. Background elements can be drawn with lower-polygon models, and ambient or background environment sounds can be represented by lower-fidelity audio.

It should be mentioned that any preloading of content is best done when players expect a pause, or when the game is not as active. Loading elements that are re-used in many game levels at the beginning of the game is a better practice than reloading the same elements at each new level. Loading content when game activity is low, such as during a pause or when a menu screen is being shown, also is a good practice. Loading new terrain while the player is crossing your game world at high speed will be far more noticeable than when the game is paused or when the player is moving more slowly.

Improving Performance in Game Code

A good game design can make a great deal of difference in your game's performance, but your game's code can also affect your game's performance—for better or for worse. By writing your game code with performance in mind, you will get even better performance from your game design.

Use reference and value types appropriately

Value types are allocated on the stack when they are declared, and continue to hold memory as long as they have scope. Value types require exactly as much memory as the data type that they are declared as:

an int value, for instance, holds 32 bits (4 bytes) of data. A long value holds 8 bytes of data. When copying a value type, the entire block of memory held by the value type is copied to a new location in memory.

A *reference* type is allocated on the heap, along with an accompanying value type that acts as a pointer to the reference, and a small amount of additional system overhead (4 bytes or 8 bytes). When assigned to another reference type, only the value of the pointer is copied—the memory held by the reference type is unchanged.

In C#, structs are value types, and classes are reference types. Because copying value types requires a transfer of all of a value's data to another location in memory, reference types are preferred for cases in which you will store a large (more than 16–64 bytes) amount of data that may be used in different places. Pass data by reference to methods when the data object is prohibitively large to pass by value.

Since reference types have additional memory overhead, they are not as efficient as value types when dealing with small amounts of data (less than 16–64 bytes). In this case, it is better to use a simple struct and pass data by value.

Note: The best way to determine whether your performance strategy is successful is by testing. Use a frame-rate counter, timer, or other metric to see how changes in your code affect game performance. Links to tools you can use to test performance are provided in the Additional Resources section at the end of this article.

Understand how Garbage Collection works on Xbox 360

C# is a memory-managed language. Unlike C and C++, the programmer does not need to explicitly allocate and deallocate memory for objects. Instead, a technique called garbage collection (GC) is used to intelligently release memory to the system when objects are no longer being used.

The .NET Compact Framework (CF) used by the Xbox 360 uses the mark-and-sweep method of garbage collection. This occurs once per megabyte of memory allocation, and involves a traversal of every object currently allocated on the heap (reference types, as mentioned previously) to determine whether they are still being referenced. All objects not referenced are freed, and all objects that are still referenced are compacted: objects are moved to sequential memory locations to reduce the total amount of memory held by the game. This is a computationally expensive operation, and can reduce your game's performance when it occurs.

To reduce the effect of mark-and-sweep operations on your game's performance, there are a number of things you can do in your code. These are:

- **Allocate memory in batches.** Since C# allocates objects linearly, objects allocated together are located one after another on the heap. This reduces memory fragmentation, which is a major source of the computational expense of a mark-and-sweep operation.
- **Use object pools.** Since garbage collection passes occur after each megabyte of memory allocations, performing fewer allocations will reduce the number of these passes. By allocating a reusable pool of objects at one time, and then using your own reference-counting strategy to

determine when to release or re-initialize an object, you can avoid frequent garbage collection passes.

- **Use fewer references.** Since value types are not garbage-collected and do not contribute to the one-megabyte allocation threshold that causes a new mark-and-sweep pass, prefer them over reference types when working with small blocks of cohesive data.
- **Initiate garbage collection strategically.** Calling **GC.Collect** initiates a mark-and-sweep pass and resets the one-megabyte allocation threshold. You can explicitly initiate a garbage collection pass in areas of your code where a number of allocations have been made, and when the player is less likely to notice a drop in performance due to the mark-and-sweep pass. An example of a good place to do this is at the end of your game's **LoadContent** method, since the code in this method typically involves a number of object allocations, and it is called before the game's update/render loop begins running. Carefully examining the various game states may help identify other opportunities for calling **GC.Collect**, such as just before the game resumes from a pause or between game levels.

Use multithreading sparingly

Multithreading on the Xbox 360 is constrained by memory, hardware, and the .NET CF. For example, thread priority cannot be set on Xbox 360. Threads introduce both memory and process overhead, require careful management to avoid deadlocks, and can reduce the overall performance of your game when used incorrectly.

Keep additional threads light by using **Sleep** and **Wait** wherever possible to enable the main application thread to use available CPU resources and to keep your game loop running smoothly. Avoid using threads where possible.

XNA Game Studio 4.0 provides many asynchronous methods for tasks that may take a while, eliminating the need for you to explicitly create threads to perform these tasks. For more information, see [Working with Asynchronous Methods in XNA Game Studio](#) on MSDN or in the XNA Game Studio documentation.

Because **ContentManager.Load** is not asynchronous, a good example of multithreading would be when progressively loading or preloading game assets where the aim is to load upcoming screen elements while still running the main game loop. To do this, define a background thread (or, preferably, use one from the [ThreadPool](#)) that loads the new assets and sets a flag when it is finished. The main thread can notify this background thread that new elements should be loaded by placing them in a loading list (note that you will need to add code to lock and unlock the list so that both threads cannot simultaneously access the list).

Avoid boxing and unboxing

As for any .NET application, avoid the expensive process of boxing and unboxing that occurs when converting value type values to reference objects and vice versa.

Use String/StringBuilder wisely

Choose between **String** and **StringBuilder** according to your game's needs and required efficiency.

System.String creates a new referenced object *every time the string is changed*. Any additional references to a **String** object point to the original, unchanged version of the string. Continuing to change **String** objects will cause allocations that can trigger a garbage collection pass and reduce game performance.

StringBuilder is a standard reference type that holds and manipulates a block of memory containing the string data. It does not create a new instance of the object when it is changed, which places fewer demands on memory and is unlikely to cause a garbage collection pass to be initiated. Because all references to a **StringBuilder** object point to the same area of memory, exercise care when using multithreading to ensure that different threads are not simultaneously changing the same **StringBuilder** object.

If your strings are relatively long lived and change infrequently, use a **String**. For strings that change often, use **StringBuilder** to avoid frequent garbage collection passes. Further gains might be obtained by pooling and reusing **StringBuilder** objects.

Avoid reflection

Reflection has substantial costs, especially in the limited .NET CF environment. Whenever possible, prefer coding with strong typing and foreknowledge of the assemblies used in your game.

Additional Resources

Performance is a large, multifaceted topic. It is impossible in one article to account for every performance-related concern for Xbox 360. There are many resources available that can provide you with even more information about how to improve the performance of your Xbox 360 games by using XNA Game Studio. For more information, see the following resources:

XNA Game Studio 4.0 Documentation

The XNA Game Studio 4.0 documentation features the following articles that discuss performance on Xbox 360:

- [Achieving Good Performance on Different Hardware Types](#)
- [Thread Pools in the .NET Compact Framework for Xbox 360](#)
- [XNA Framework Remote Performance Monitor](#)

App Hub

The [App Hub](#) contains tools, samples, and educational content related to performance. See the following areas for details:

- The [Performance Education Catalog](#) provides articles, samples, and tools to help you with your game's performance.
- [XNA Game Studio Forums](#) are a great place to ask questions about performance and to get them answered quickly by members of the XNA Game Studio development community and by the developers and testers at Microsoft who build XNA Game Studio!

MSDN Blogs

- [Shawn Hargreaves' Blog](#) is an excellent source of performance-related articles specific to XNA Game Studio.
- [Nick Gravelyn's Blog](#) also contains many articles that focus on performance.