Optimizing performance of XNA on Xbox 360

Nikolaj Leischner, Olaf Liebe and Oliver Denninger

FZI Research Center for Information Technology

http://zfs.fzi.de

The initial situation

This article provides an insight into the process of performance optimization for Xbox 360. We cover some tricks and pitfalls that are specific to the current version of XNA. Our observations are based on the code base of a game, created by students during a project course at the University of Karlsruhe. Due to limited course time the result was a functional but performance-hungry game – too demanding for the Xbox 360. We managed to achieve average frame rates 5 times higher than before, with moderate effort – enabling smooth game play on Xbox 360.

In the game the player controls a hen with the ability to walk and jump through a 3D world, seen from a rotatable 3rd person camera. The game world is a height-map based terrain with various 3D objects. The player’s motion is limited by these objects and the borders of the terrain.

The player’s goal is to discover a number of hatchlings, which will follow him once they are found, and to bring all of them back to the hen house. To discover the hatchlings, the player has to solve small puzzles based on scripted behavior of non-player objects, e.g. one hatchling has to cross a bridge which is blocked by a cat standing on it. A nearby dog can be lured to the bridge where he will chase the cat away.

Things to consider

When looking at performance on Xbox 360, you should consider:

- **Garbage collection:** The .NET Framework on PC uses a generational approach, making garbage collections less painful and more resilient to large numbers of objects. With 512 MiB of GDDR3 memory shared between GPU and CPU the Xbox 360 garbage collector can't afford such luxury.

  The garbage collection of the .NET Compact Framework for the Xbox 360 always has to check all objects. Therefore the time a collection takes (called the garbage collection latency) increases linearly with the number of objects. Further a collection will be triggered whenever 1 MiB of memory has been allocated. So we need to allocate as much as possible up front and keep the number of objects small (as a rule of thumb below 100,000).

  The first version of our game didn't consider this, which results in very annoying stuttering. Note that this stuttering is independent of the frame rate. It is caused by the garbage collection and not by slow frame calculations! If a game is very wasteful in terms of memory allocation it could run at 300 frames per second and still be unplayable on the Xbox 360. The same game might run on a PC with no trouble at all.

- **The compiler and the PowerPC CPU:** On the one hand the .NET compiler on the Xbox 360 isn't (yet) very good at code optimization. Manual tweaking can make a big difference. On the other hand a single core of the Xbox 360 CPU is rather comparable to a low-end PC CPU plus we currently don't get its full performance with .NET and XNA. That is because XNA doesn't currently use the AltiVec[1] instructions provided by the CPU for vector operations. Note that this may change (as many other details mentioned here) with future versions of the XNA framework. This isn't too bad. But it means that the Xbox 360 is more vulnerable to inefficient code and that some ugly optimization can be very beneficial. If you use threads, it's possible to take advantage of up to four of the six cores of the Xbox 360 CPU. But we do not use threads.
- **XNA 2.0:** The XNA framework is by no means final at this time. Some functions are implemented in an inefficient way, leading to unexplainable performance characteristics.

There are two GDC talks discussing these topics, both worth listening to[2].

Our first steps

First of all we reworked parts of the code base to be more maintainable. One should be satisfied with the general design before starting to optimize details. Reworking code is a highly project-specific task. For example in our case we found that several parts of the game did a lot of local object querying, e.g. for collision detection. So we drew the conclusion to put our scene into a custom quad-tree data structure to provide a faster lookup.

To our surprise the first quad-tree implementation proved to be slower than using a simple list of objects. It took a fair bit of testing and tweaking until the results were satisfying. Testing and benchmarking is vital, as shown by this example. Just because a data structure or algorithm in theory is "obviously" more efficient, reality can prove otherwise (especially for small sets of data). But with some careful tweaking we were able to avoid some initial pitfalls, making the quad-tree faster in the end.

After completing code refactoring, performance already had increased and stuttering wasn't as bad as it had been before – but it was unfortunately still very noticeable. At this time using additional tools became essential:

- **CLR Profiler[3]:** Running an application with this tool records detailed statistics about where memory has been allocated during execution. While it has to be used on PC the hot spots are the same on the Xbox 360.
- **XNA Framework Remote Performance Monitor[4]:** Shows how much memory gets allocated per second, how much memory is in use, how many objects are alive and how often a garbage collection is performed. Taking a look at these numbers shows where object allocation has to be optimized.
- **3rd party profilers:** There are several decent profiling tools available that provide additional information about performance bottlenecks. Using such tools isn't

1 http://en.wikipedia.org/wiki/AltiVec
2 http://www.microsoft.com/downloads/details.aspx?FamilyId=B11AD912-4158-44CC-A771-A5E044F7E3BB&displaylang=en
http://www.microsoft.com/downloads/details.aspx?FamilyId=8450DB46-283F-4924-B35C-3CCD1DB7E97E&displaylang=en
3 http://www.microsoft.com/downloads/details.aspx?FamilyId=A362781C-3870-43BE-8926-862B40AA0CD0&displaylang=en
4 http://msdn.microsoft.com/en-us/library/bb975830.aspx

essential; furthermore they cost some money.

- **In-game statistics:** Measuring in-game time can also be useful, to track performance issues, without the need of using a profiler.

For our game we discovered that it was allocating as much as 8 MiB per second while using approximately 140,000 objects. Remember 8 MiB of allocations per second cause 8 garbage collection runs per second. Unfortunately we couldn't hope to decrease the number of objects that much, without changing a lot of the source code (especially since a lot of objects were created by the external script interpreter, which we didn't want to modify). Therefore we tried to assure that collection happens rarely.

We will present some tricks and techniques which we used to reach this goal.

Reducing object allocation

Use structures instead of classes

There are two benefits of structures compared to classes for game development on the Xbox 360: first, as they are allocated on the stack and not on the heap, allocation is almost for free (only the stack-pointer has to be increased and the constructor executed – if available). Second, garbage collection does not need to keep track of them, reducing the total number of objects handled by garbage collection.

However, structures have their drawbacks: passing them as arguments to methods involves copying the whole structure. Furthermore structures always need a parameter-less default constructor (which you sometimes may want to avoid due to encapsulation). And more important, structures can lead to errors, which are often hard to find, for people used to the behavior of classes.

You can avoid the fact, that structures are always copied by using the **ref** and **out** keywords as described later. But note that this imposes different problems and makes the code a bit awkward to read. Another risk, we often ran into, comes with the use of **foreach**. When iterating an array with **foreach** or with an enumerator in general, you actually only get a copy of the underlying structure, because structures are always copied. First, this copying needs additional performance and second, changing an object during iterating won't change the object in the iterated data structure as often intended. Furthermore it's not possible to pass the currently iterated object using **ref**, which is fortunately detected by the compiler. In such a case it is often better to use a simple **for** loop, which avoids those pitfalls, but makes the code harder to read and requires a data container that allows direct access by index.

In our case, we initially stored the keyframe data of the models animations using classes. Switching to structures, we were able to reduce the total number of objects dramatically (there are a lot of keyframe objects created), thus reducing the garbage collection lag. Another example is the data structure returned by our collision tests, to provide further information about a collision. This occurred multiple times per update, always creating a new object for garbage collection. Using a structure, we got rid of that with almost no effort.

As a rule of thumb you should always consider structures for small data objects, that are often created or destroyed, or objects that occur in great numbers and just lie around all the time, imposing unnecessary load to the garbage collection. But especially objects created multiple times during a frame or update are good candidates for structures. Note that the CLR Profiler can immediately show you, how much allocation you saved exactly.

Re-use frequently used objects

Re-using objects is a good idea in general as it reduces the number of garbage collections. On the other hand, trying to avoid object creation *at all costs* reduces readability and maintainability of your code. However, there are some classes in XNA which have large objects, an expensive constructor or both – for such objects re-using obviously can be very beneficial.

The **BoundingFrustum** for example is particularly expensive to create (lots of arithmetic operations, for decomposing the matrix, calculating plane intersections, normalizing vectors) **and** to destroy (as it contains about eight arrays, even some multidimensional ones).

The **SpriteBatch** also belongs to these objects. In our case, re-creating two **SpriteBatches** on a per frame basis degraded performance to 1/3 in comparison to creating them once and then re-using them.

Re-use allocated lists instead of creating them anew

Another important example for re-usage is **List<T>**. We used to create lists in several update methods, most notably to store objects returned by queries to the quad-tree (for example to trigger an action, when the player approaches some other object). To use a list in such a way is a bad idea, as creating and filling the list will create several objects. That is because the list uses a static array with an initial capacity internally. When more capacity is needed, a larger array will be created (twice the size, although that might change) and the data will be copied to the new array, leaving the old array for garbage collection. For illustration, consider the following situation:

You create a list and fill it with 20 objects. The list is initialized to a capacity of 4 elements. The next array has 8 elements, the one after that 16 and so on. While filling the list, we created four arrays. With the list itself, that makes five objects that need to be collected!

You can avoid that by extracting the list declaration from the method and clearing the existing list instead of creating a new one each time. This way the lists capacity stays where it is, so that the list fills up once to the capacity needed and then just stays there. That behavior is even documented, so it is unlikely to change.

Again use the CLR Profiler, to watch the effect. Though, keep in mind that you have to run the game for some time to note a difference, as the initial allocation won't diminish.

Avoid creating arrays frequently

As every array is an object, arrays cause additional load for the garbage collection. Creating (small) arrays within inner loops can be fatal for this reason. So if one needs e.g. 6 floats which would semantically form an array it might be a lot faster to create 6 different float variables instead. Calling a method with a parameter list specified using the **params** keyword (like **String.Format()**) has the same drawback. But again, avoiding arrays leads to more error-prone code and shouldn't be done excessively.

Avoid boxing and unboxing

Boxing refers to the technique of embedding a value-type, such as an integer into an object. On the one hand, this enables you for example, to put an integer into an array of the type **object[]**. But on the other hand, boxing creates a new object each time. This object will transparently replace the actual value, until you cast it back to its value-type.

Unfortunately, like any other object, these little wrapper objects have to be handled by garbage collection. So if you still have problems with garbage collection, you might take a look at that.

The best way to find boxing and unboxing in your source code is to disassemble the executables and take a look at the CIL code. To do this you can use **ildasm.exe**, the disassembler of the .NET Framework SDK. Search for it in your Visual Studio directory under "**SDK/.../Bin**" or below the Program Files directory in "**Microsoft SDKs/Windows/.../Bin**" or "**Microsoft .NET/SDK/...**". It should be somewhere among those directories.

After you started ildasm and opened your binary, you can export the CIL code using File → Dump. Search the exported file for **box**, **unbox** and **unbox.any**, to find boxing and unboxing commands. CIL code is hard to read, but normally it's sufficient to figure out the name of the method where boxing or unboxing occurs. Then you can inspect this method in the source code.

Improving execution speed

Note that most of the following tricks have severe drawbacks beside their advantages. Use them with care and consider where to use them exactly. Some may introduce side-effects that can be hard to track down, while others degrade the code's readability and maintainability. Use a profiler to find bottlenecks in your code and optimize only those. Test the effect of those optimizations on the execution time, to see how much optimization is actually needed. Premature optimization will most likely only make things worse, with small effect on the actual speed.

Use ref and out for value-types

Passing matrices and vectors can be quite expensive for methods that are frequently called. As any other value-type (and this trick applies to all of them), passing them always copies the data to the stack. So especially for small methods and large value-types (such as a matrix) that often means, that most of the time needed for the method is actually used to copy arguments or results.

This can be avoided by using the **ref** keyword that will only pass over a reference to the value-type, enabling the method to access the data directly. A similar solution is available for return values with the **out** keyword. It allows you to specify a variable to store the solution up-front; so that the data can be directly copied to the place it is required.

Note though, that those advantages don't come without drawbacks. As you operate on the same data structures, changing them will also change the value outside of the method. Such side-effects are especially hard to find, so be careful when using arguments passed by reference!

This trick is even used by the XNA framework in many places. For example there is not only a version of **Matrix.CreateLookAt()** that returns a matrix, but also one that allows you to specify a matrix to fill, using **out**. The same applies to **Matrix.CreateRotationX()**, **Matrix.CreateScale()** and many other methods. Look them up in the documentation.

Be careful with operators on vectors and matrices

As you most likely know, the vector and matrix classes provided by the XNA framework have overloaded operators. That allows you to multiply matrices A and B, by just writing "**A * B**", which is pretty nifty. Internally, this will actually call a static method of the matrix structure and pass the arguments A and B by value, effectively copying them and the result using the stack. So using the above multiplication will copy over three matrices, which can be quite expensive sometimes.

Fortunately, the XNA framework provides an alternative: there is the method **Matrix.Multiply()**, which effectively does the same multiplication, but allows the arguments to be passed as references. Similar methods exist for the other arithmetic operations on matrices **and** on vectors.

The disadvantage is of course, that the resulting code is less readable. So consider the use for time-critical methods mainly. Use a profiler to find them and examine the results.

Unroll loops manually

The current compiler is pretty limited, concerning simple optimizations. Loop unrolling is an example of that. If you have an array of fixed size (about 5 elements) and perform an operation on each of them using a loop, consider writing five instructions instead. The reason is, that usually executing a loop will result in the execution of the included code, a comparison for the stop criterion, a jump back to the beginning and so on. So simply writing five instructions will eliminate the comparison and jump instructions.

We expected the compiler to be smart enough to unroll all those simple loops and used them in the code that detects collisions between two bounding boxes, to iterate the three orientation vectors that each bounding box contains (pointing from the center to each side). It really astonished us that eliminating the loops manually gave some speed increase.

Note that the compiler will likely become more optimized in future versions of XNA and that unrolling loops manually may produce code that is much harder to read. So you should really reduce this technique to the methods that are an absolute bottleneck.

Know the do's and dont's of XNA

There are several ways to do things in XNA, and some of them aren't fast. We will enumerate a few common things that are better not done. The referenced GDC talks elaborate on some of these in more detail and mention further pitfalls.

1. Reading render states is slow, don't do it unless you have to. Simply set the parts you want to change, regardless of their current state.
2. The cost of **SpriteBatch** greatly depends on which settings one uses. Never set them to save the render state and if possible use the immediate sort mode.

3. Even though it seems convenient: for effects it is not recommendable to access their parameter collection for frequently changing parameters. Use **EffectParameter** objects instead.

Conclusion

In summary, our optimization efforts led us to a game running significantly smoother on Xbox 360. With basic understanding of the structure and concepts of the .NET-Framework and XNA many bottlenecks of the Xbox 360 can be bypassed. If the characteristics of the Xbox 360 had been taken in consideration during design time of the game, the result should be even better.