Bachelor’s thesis

Information and Communication Technology

2023

Konstantin Ionin

Comparing performances between different methods of using large numbers of ParticleSystem effects in Unity games on Android OS


TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communication Technology

2023 | 48 pages

Konstantin Ionin

Comparing performances between different methods of using large numbers of ParticleSystem effects in Unity games on Android OS

Unity game engine makes creating new entities in games easy at runtime with its Instantiate method. However, rapidly creating short lived objects results in more memory management tasks for the CPU. One method of reducing the frequency of allocating and releasing memory is the object pool pattern, also referred to as pooling, which consists of allocating the required memory in advance and reusing allocated objects rather than allocating memory for a new object each time and releasing it later.

The purpose of this thesis was to determine how pooling reusable particle effect objects can affect performance, compared to allocating new objects as needed with Unity’s Instantiate method, on smartphones running Android operating system, particularly on older devices. Testing was conducted with several Unity applications that would create various amounts of particle effects on the screen with these methods, and logged time and memory use during each frame over a period of time, and results were then analyzed and compared. The results indicate that even with frequent use, pooling may not be significantly faster and mostly affects memory use. However, more detailed profiling may be needed with specialized tools.

Keywords:

object pool pattern, memory, RAM, garbage collection, GC, Unity, Android.

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

Opinnäytetyön valmistumisajankohta | 48 sivua

Konstantin Ionin

Suorituskyvyn vertailu erilaisten ParticleSystem efektien käyttömenetelmien välillä Unity peleissä Android käyttöjärjestelmällä

Unity-pelimoottori mahdollistaa uusien olioiden helpon luomisen pelin ajon aikana Instantiate-metodin avulla. Uusien olioiden luominen tiheään tahtiin ja lyhyeksi ajaksi toisaalta lisää muistin käsittelyn työtä prosessorille. Yksi tapa vähentää muistin jatkuvaa varaamista ja vapauttamista on object pool pattern -eli pooling -menetelmä, jolloin varataan tarvittava muistimäärä ja luodaan tarvittavat tietorakenteet etukäteen, minkä jälkeen niitä käytetään toistuvasti uusien luomisen ja tuhoamisen sijaan. Tämän työn tarkoitus on selvittää miten uudelleenkäytettävien particle-efekti -olioiden käyttö voi vaikuttaa sovelluksen suoritukseen, uusien olioiden luomiseen ja tuhoamiseen verrattuna, Android-käyttöjärjestelmää käyttävillä älypuhelimilla, varsinkin hieman vanhemmilla laitteilla. Menetelmien testausta varten oli toteutettu useita samankaltaisia sovelluksia Unity-pelimoottorilla, joissa ruudulle luotiin suuria määriä particle-efektejä. Sovellukset kirjasivat ajan ja muistin käytön jokaisella framellä tietyllä aikavälillä, minkä jälkeen tuloksia verrattiin ja analysoitiin. Tulokset viittaavat siihen, että pooling ei välttämättä nopeuta suoritusta suurellakaan olioiden määrällä ja vaikuttaa ensisijaisesti muistin käyttöön, mutta kattavampi tutkimus erikoistuneilla työkaluilla voi olla tarpeellista.

Asiasanat:

object pool pattern, muisti, RAM, GC, Unity, Android

Contents

Figures

Tables

List of abbreviations

Abbreviation	Explanation of abbreviation (Source)
abd	Android Debug Bridge (Google 2023b)
CPU	central processing unit (Lemonaki 2021)
GC	Garbage Collector / Garbage Collection (Microsoft 2021)
OS	operating system (Finto 2018)
RAM	Random-access memory (Oxford University Press 2023)

1 Introduction

In many games there is a need to create a temporary visual effect in response to some player action or an event. Unity facilitates this with the Instantiate method (Unity Technologies 2022d). If events that require temporary objects happen rapidly, frequent memory allocation and then garbage collection can create significantly more work for the central processing unit (CPU) of the device that runs the game or application (Unity Technologies 2023d).

This seems to be a particularly important issue on mobile devices because of more limited random-access memory (RAM), battery capacity and processing power.

This study was inspired by development work at Pikkuli Group Oy, with newfound interest in memory use optimization.

The structure of this thesis is as follows: Chapter 1 is the introduction to the goals and structure of this study. Chapter 2 briefly explains managed memory and how it's relevant to this study. Chapter 3 describes the object pool pattern in detail and presents its general advantages and drawbacks. Chapters 4 explains the structure of test applications and Chapter 5 describes how the tests were conducted. Chapter 6 presents the test results and chapter 7 describes their implications.

The object pool pattern is a design pattern and a memory optimization method for reducing the frequency of memory allocation. (Nystrom 2021; SourceMaking 2023; Koulaxidis et al. 2022; Bonet 2021.) It has potential drawbacks, such as requiring allocation of large amount of memory for an extended period and generally requiring objects to be reset or otherwise modified between uses, which also adds different tasks for the CPU. (Nystrom 2021; Bonet 2021.) This design pattern is not new but can be easily overlooked and takes more effort to implement in Unity scripts than just instantiating new objects as needed.

The goal of this study is to determine how pooling ParticleSystem objects can affect the performance of mobile games, when it may be better to pool them

and when it may be better to not pool them. Having concrete results could benefit mobile game developers in general, but particularly those who use the Unity game engine.

This study focused on older and low end to non-high end mobile devices because those would be more easily affordable to more people and are still in use, and because high end ones would be more likely to run even inefficient software well. Similar tests on more powerful mobile devices may be worthwhile.

The methodology consisted of running different variations of an application that would create numerous particle effects on the screen, either by creating new objects in memory or by using a pool of objects, and logged specific performance metrics, that could then be compared and analysed. The metrics recorded were total memory used by specific objects, including particle effect objects, total memory reserved by the application, the number of particle effect objects, the time between start of the frame and the main task (thread) finishing the job, and the time elapsed from the start of the game at each frame. The time elapsed was also translated to total times between frames. Reusable objects differed from non-reusable ones only by not destroying themselves automatically and by having an additional component that allowed the object pool to reuse them.

2 Managed memory in Unity, C#, and Android OS

In C-like languages, including C#, the heap is a portion of memory from which a program can reserve some amounts for its use in addition to the memory reserved automatically for the program code (Albahari et al. 2015a; Chen et al. 2020). Unlike the memory that is automatically reserved for the program during its execution, memory reserved from the heap by the program is not automatically freed and remains reserved until the program is closed (Chen et al.). Programmers can write instructions for when to free specific parts of reserved heap memory as part of their code, like in C++ language, or they can rely on an automated process that tracks which parts of heap memory are used and frees the parts that are not used anymore. An automated method of freeing unused memory is called garbage collection, and the process or the implementation is called a garbage collector, abbreviated as GC. (Chen et al.; Microsoft 2023b; Wienholt 2004.) Garbage collection can be a feature of a programming language as is the case with C#, implemented in a code library for a language that developers can use, feature of a game engine like Unity, or even a feature of an operating system as is the case with Android. (Microsoft 2023b; Boehm et al. 2014; Unity Technologies 2023e; Google 2023g)

2.1 Android memory management.

Android tries to use all available memory, keeping apps in memory after they are closed, so that the user can quickly switch to them if needed (Google 2023f).

If an application reserves memory for itself, it generally can’t be released by Android unless the application releases it by itself, the application has mechanisms for Android to reclaim parts of the memory used by that application, or Android decides to kill the process to use its memory for something else. The latter may happen if available memory is scarce and the application in question is not on the foreground (Google 2023f; Google 2023f).

Android has generational garbage collection, meaning objects that are recently allocated are expected to have shorter lifespan, and are checked more often, whereas objects that have existed for longer are expected to last and are checked less often (Google 2023g).

2.2 C# and Unity memory management

In C#, a managed heap refers to a portion of memory reserved by the garbage collector for any temporary memory allocations (Microsoft 2023d). The purposes of garbage collection and a managed heap are to reduce the need to write memory management code, and to make memory allocation more efficient and safer (Microsoft 2023b).

C# language, like Android OS has generational garbage collection. Dynamically allocated objects that are not released for a longer time are then checked less often, so the garbage collector can focus on objects that are more likely to be released soon (Microsoft 2023c).

Unity has a managed memory system, in a form of C# scripting environment, that manages the releasing of dynamically allocated memory, so that developers don’t have to do it manually. Unity can expand the managed heap to accommodate the need for more memory. If managed heap expands regularly, Unity won’t release the memory allocated for it, even if most of it is not used. (Unity Technologies 2023f.)

This implies that Android’s garbage collection is not as significant for memory use of a Unity game / application and Unity engine handles temporary objects itself unless it needs to reserve more memory or decides to release some of it. In some ways, a managed heap can be thought of as a memory pool.

By default, Unity games will use incremental garbage collection, which means that the GC will split its workload over multiple frames, avoiding the need to pause other tasks to process all the reserved memory (Unity Technologies 2022c). Test applications used this setting because it is the default and is recommended.

2.3 Memory fragmentation

When memory is allocated from the heap, it is done in contiguous region. When that region of allocated memory is released later, there may be other parts of memory adjacent to it that are still reserved, and as a result the released portion of memory can only be used for allocations that are small enough to fit in that free segment. This is known as memory fragmentation. Even if there is free memory in the managed heap, it cannot be used for allocations that are larger than any of the free segments. (Nystrom 2021; Unity Technologies 2023f.) This could lead to Unity having to expand its managed heap (Unity Technologies 2023f). However, according to C# documentation, managed heap and garbage collector will move live objects closer together during garbage collection process to avoid this (Microsoft 2023d). Unity generally uses C# scripting environment (Unity Technologies 2023f), so memory fragmentation may not be a significant issue.

3 Object pool pattern as an optimization method

As mentioned in the introduction, object pool pattern, or pooling, is a design pattern / memory optimization method that is used to reduce frequency of dynamic allocation and freeing of memory by allocating the required memory in advance and initializing a set number of reusable objects, referred to as an object pool. Processes can then retrieve and return these objects as needed instead of creating new objects and then destroying them.

Main trade-off is that the memory is not released for a longer time, and the pool has to be able to accommodate the number of objects that can be used at a time. This can mean reserving a lot of memory and not freeing it even when significant portion of it is not used, depending on how pooled objects are used. (Nystrom 2021; Bonet 2021.)

Different languages and game engines allow for different implementations of object pools. Unity has its own built-in generic object pool interface IObjectPool<T0>, and classes ObjectPool<T0> and LinkedPool<T0> that implement it. (Unity Technologies 2023g.)

Improvements to garbage collection have led to object pools being considered unnecessary, and in some cases even detrimental in some situations (Goetz 2005).

4 Test applications and devices

The application used to compare performance of continuous creation and destruction to that of pooling was a simple Unity application with one scene and scripts for a few classes. Unity version used for this was 2022.1.22f1.

4.1 Important Unity application and programming terms

GameObject

GameObjects can represent various props, scenery and characters. Their functionalities are defined by components attached to them. GameObjects can be enabled and disabled, which also enables and disables functionalities of their components, respectively. (Unity Technologies 2022a.)

ParticleSystem

ParticleSystem component can be attached to GameObjects to create particle effects in games / applications (Unity Technologies 2022f).

ParticleSystem objects in application description will refer to GameObjects with ParticleSystem components attached. A pooled object would consist of a GameObject with a ParticleSystem component and a simple component that allowed it to interact with the object pool.

Prefabs

Unity has a way of saving templates of GameObjects and groups of GameObjects with various components as reusable assets known as prefabs (Unity Technologies 2022g). The applications used for testing relied on prefabs that consisted of GameObjects with ParticleSystem components.

Thread

In computing, threads are execution paths that can run concurrently with each other and independently of each other (Albahari et al. 2015b; Duffy et al. 2013).

4.2 Important application classes

In Unity, custom classes can be attached to GameObjects as script components (Unity Technologies 2022e).

UML diagrams do not fully describe the classes, and focus on more important variables.

ParticleSpawnRequester class requested particle effects in form of GameObjects with ParticleSystem components from one of two different classes, ParticleSystemLibrary and ParticleSystemPool, depending on settings (Figure 1). Both classes used prefabs to create new objects as needed. ParticleSpawnRequester had a timer to keep requesting particle effects for a specified duration and could request a specific number of particle effects at a time. The tests were done with 1 and 3 effects at a time.

In addition to controlling the use of ParticleSystems, ParticleSpawnRequester alsoperformed a slow calculation on each frame to simulate the application having other tasks in addition to causing particle effects on the screen.

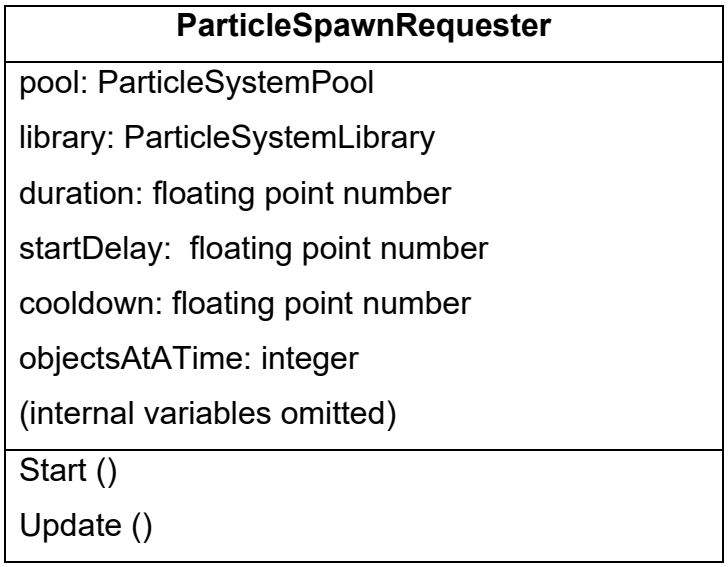

Figure 1. ParticleSpawnRequester class diagram

ParticleSystemLibrary class would use Instantiate method to create new game objects on request, which would be destroyed after performing their function, allowing the memory to be released by the garbage collector (Figure 2). ParticleSystem components were set to play when created and to destroy GameObjects they were attached to after their set duration.

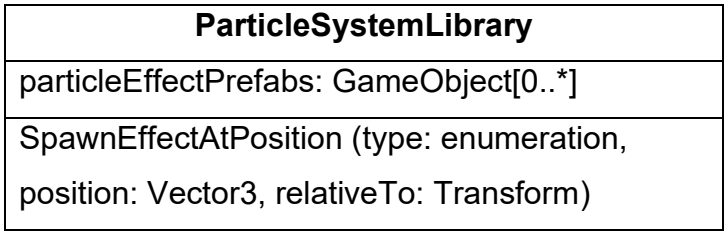

Figure 2. ParticleSystemLibrary class diagram

ParticleSystemPool class maintained an array of object pools. The reasoning for an array of pools was that this could be a usable implementation for a game with several particle effects that could all be pooled (Figure 3).

ParticleSystemPool
pools: ParticleSystemObjectPool [0..*]
Start () SpawnEffectAtPosition (type: enumeration, position: Vector3, relativeTo: Transform)

Figure 3. ParticleSystemPool

Each object pool was an instance of a ParticleSystemObjectPool class that contained an instance of Unity's generic ObjectPool<T0> class and methods that this class instance (object) could use (Figure 4). Reasons for using ObjectPool<T0> objects are their ability to grow to accommodate higher number of pooled objects, up to a limit, and being a built-in feature with provided examples, making them likely to be used by newer developers. Pooled game objects also had a script component for interactions with object pool that managed them.

ParticleSystemObjectPool
prefab: GameObject poolObjectContainer: Transform minPoolSize: integer maxPoolSize: integer pool: ObjectPool<GameObject>
InitializePool () CreatePooledItem (): GameObject OnReturnedToPool (GameObject) OnTakeFromPool (GameObject) OnDestroyPoolObject (GameObject)

Figure 4. ParticleSystemObjectPool class diagram

LargeAllocationSimulator class was added after initial tests to add larger memory allocations, to test cases where memory fragmentation was presumed by the author to be more likely due to more varying sizes of memory allocations (Figure 5).

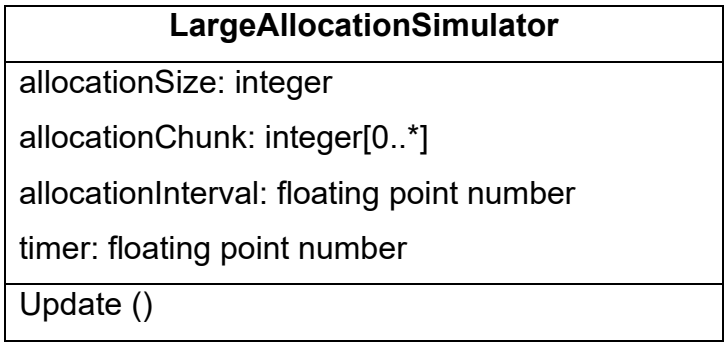

Figure 5. LargeAllocationSimulator class diagram

Lastly, the CustomProfiling class recorded Unity's total reserved memory size, memory used by objects, including the memory that was no longer used but not yet released by garbage collector, using GetMonoUsedSizeLong method, and numbers of objects for each frame, as well as the time CPU spent executing Unity's main thread job and total time elapsed at each frame (Unity Technologies 2022b). Data was recorded for each frame to arrays of arrays that were allocated at the start of the application run time and the recorded data was written to a file after all the data was recorded. (Figure 6.) Reason for using arrays of arrays was to possibly reduce the chances of not being able to reserve a large enough single array, although this might not have been necessary. Memory allocation was done at the start to avoid additional memory allocations during the tests that could affect the profiling.

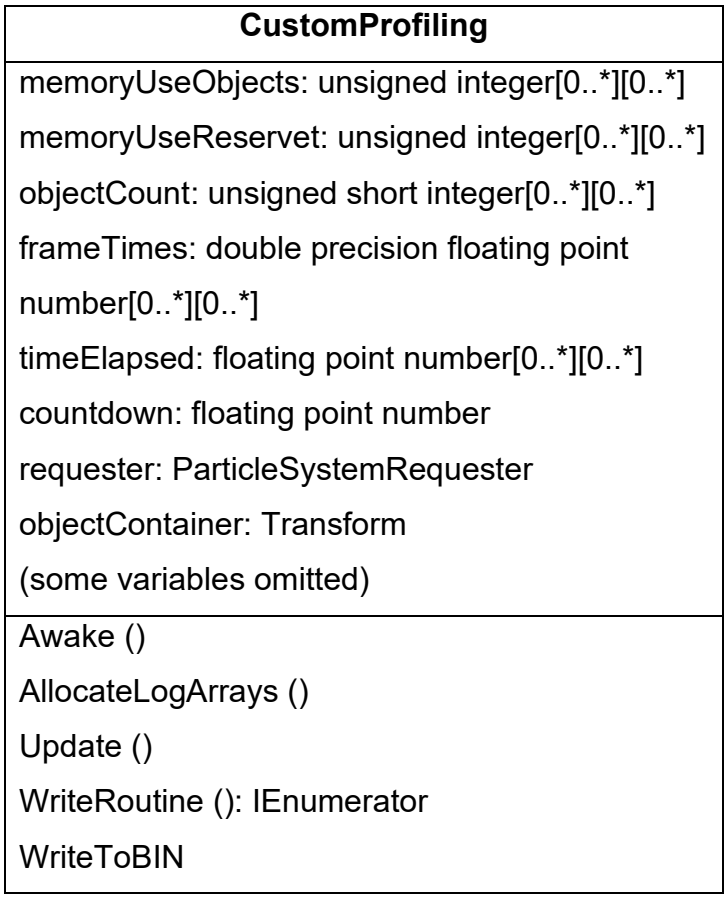

Figure 6. CustomProfiling class diagram

The API used for retrieving CPU main thread times per frame was FrameTimingManager. Using it required enabling the Frame Timing Stats option in Project Settings, or Player Settings creating a build of the application. It is primarily intended for use with Dynamic Resolution option and required that feature to be enabled in the main camera. (Unity Technologies 2023c.)

4.3 Device information

The devices used for testing were Samsung SM-J500FN and Samsung SM-A600FN/DS (Table 1).

Table 1. Tested devices and their specifications.

Device (Model number)	**Samsung Galaxy J5** (SM-J500FN)	**Samsung A6** (SM-A600FN/DS)
CPU	Quad-core 1.2 GHz Cortex-A53	Octa-core 1.6 GHz Cortex-A53
RAM available (approximately)	450 MB	1.3 GB
Android version	10	6.0.1

Model numbers and OS versions were taken from devices' "About device" and "About phone" sections in their settings menus. Available RAM was taken from Smart Manager and Device care sections in options menus of Samsung Galaxy J5 and Samsung Galaxy A6, respectively. CPU information was taken from GSMARENA based on phone model numbers. (GSMARENA 2023a; GSMARENA 2023b)

Due to the number of charts produced during testing, Samsung Galaxy J5 test results are not included in this document, but are available on GitHub along with the Unity project.

4.4 Data recording

Arrays for recording data were allocated at the start and sizes are kept the same for all versions of the application to keep their memory uses as close to

each other as possible during each test. The data in the arrays was written to a binary file after each test (Microsoft 2023a).

The log files consisted of series of binary representations of recorded values in the same order for each frame (Table 2).

Table 2. Measurements taken and data types they are recorded as in binary files.

Measurement	**Data type**
Total memory used by game objects	unsigned 32-bit integer
Total memory used reserved by Unity	unsigned 32-bit integer
Number of ParticleSystem objects	unsigned 16-bit integer
CPU main thread frame time	64-bit floating point number
Time elapsed from application start during this frame	32-bit floating point number

This pattern could then be simply translated into rows of a comma-separated values (CSV) file (Digital Preservation Home 2021). The time elapsed from application start was also translated to time between frames in the spreadsheets.

The translation to CSV was done with a simple C# program but could also be done program or script that would read binary data according to the specified pattern. The CSV files were then turned into spreadsheets.

Applications were tested with both simple ParticleSystems that just emitted a burst of particles, that would be cheaper to create, and ParticleSystems with more complex behaviours, like changing particles' velocities and colours, that would be more taxing for the CPU to create.

4.5 A bug with use of FrameTimeManager for recording

CustomProfiling class had a bug with how CPU main thread times were logged. FrameTimeManager retrieves data with a set 4 frame delay (Unity Technologies 2023c), which lead to an erroneous solution, leading to times being delayed by additional 3 frames. This was corrected for in spreadsheets by removing first 7 values in the CPU main thread time column and shifting the remaining values in that column up.

5 Testing protocol

5.1 Connecting devices and installing test applications

Developer Options were enabled on the smartphones used for testing, including allowing installing without verification over a USB cable from the development machine (Google 2023c). A connection was then established using Android Debug Bridge, or adb (Google 2023a). Development machine was a desktop computer running Windows 10.

To disconnect devices, command “adb kill-server” was used through Windows’ Command Prompt before ejecting a connected device.

The app was built and installed and launched on the devices over USB cable with “Build And Run” option in build settings window. The test application versions were launched and left to run the tests for at least 3 times including initial launch as part of installing over USB connection, after which the log files were copied from the device. Test application closed automatically at the end, but remained in memory in the background, and was removed from background between running tests to make sure test runs were as similar as possible, although this may not be necessary.

5.2 Test application configurations

For non-pooling tests, variables were ParticleSystem object creation quantities and created ParticleSystem object types.

For pooling tests, pooling versions were made to match each non-pooling version, with both minimal sized pools and larger than necessary pools, to test both optimally sized and larger than necessary object pools affect performance, but it turned out that ObjectPool<T0> objects would not create more pooled objects than they needed to accommodate the demand for those objects.

Each configuration was tested both during first 20 seconds of run time and as a separate test for 20 seconds after 80 seconds of running.

Additionally, repeat tests were done with other mock memory allocations at regular intervals to increase chances of memory fragmentation due to varying allocation sizes.

6 Test results

Due to a large number of charts of recorded data, only some charts are presented here, focusing on tests done on Samsung Galaxy A6. Spreadsheets with data and charts are available together with the project used at GitHub (Ionin 2023).

One important thing to that became apparent about Unity's ObjectPool<T0> class is that it fills over time as pooled objects are requested and does not fill more than needed, which does introduce some potential new created objects after some application run time and does not accommodate the intended tests with too many pooled objects, making those tests runs effectively the same.

The charts below show total frame times over application run times as a green line, and time between start of the frame and when main thread finished the job for that frame as a blue line.

6.1 Rapid use of simple PaticleSystems

With simple ParticleSystems, use of object pool pattern may or may not show improvements to performance or reduction in memory use by the application, even when ParticleSystem objects are created at a high rate, in this case around 10 objects per second.

In the case of an application that creates new objects to use for approximately 20 seconds we see fluctuating number of objects over time and increase of used memory. (Figure 7; Figure 8).

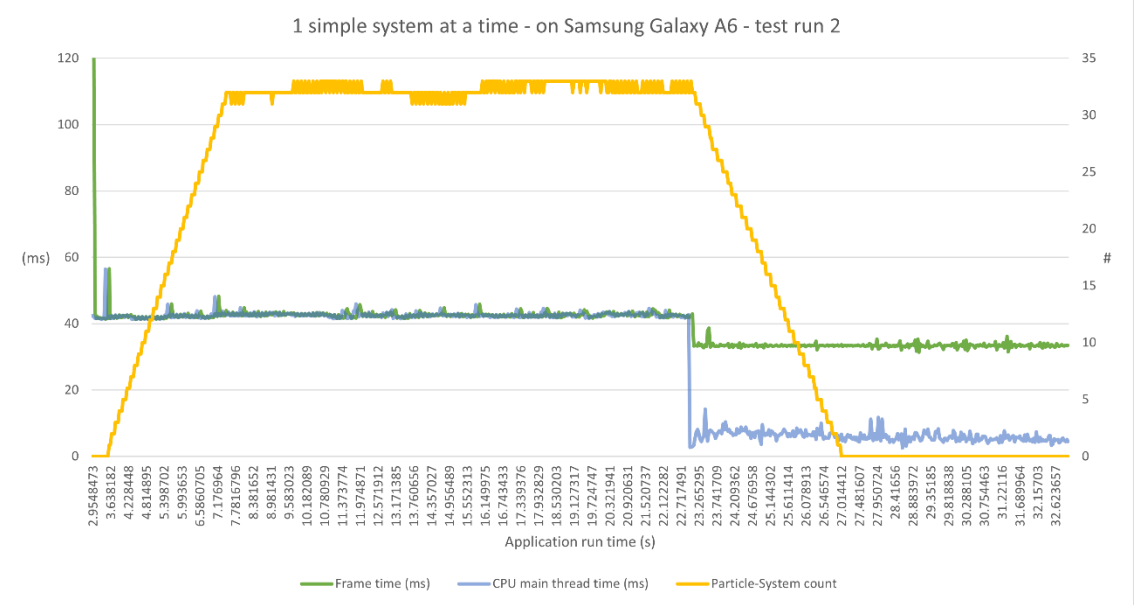

Figure 7. ParticleSystem count over 30 approximately seconds, without pooling, simple systems.

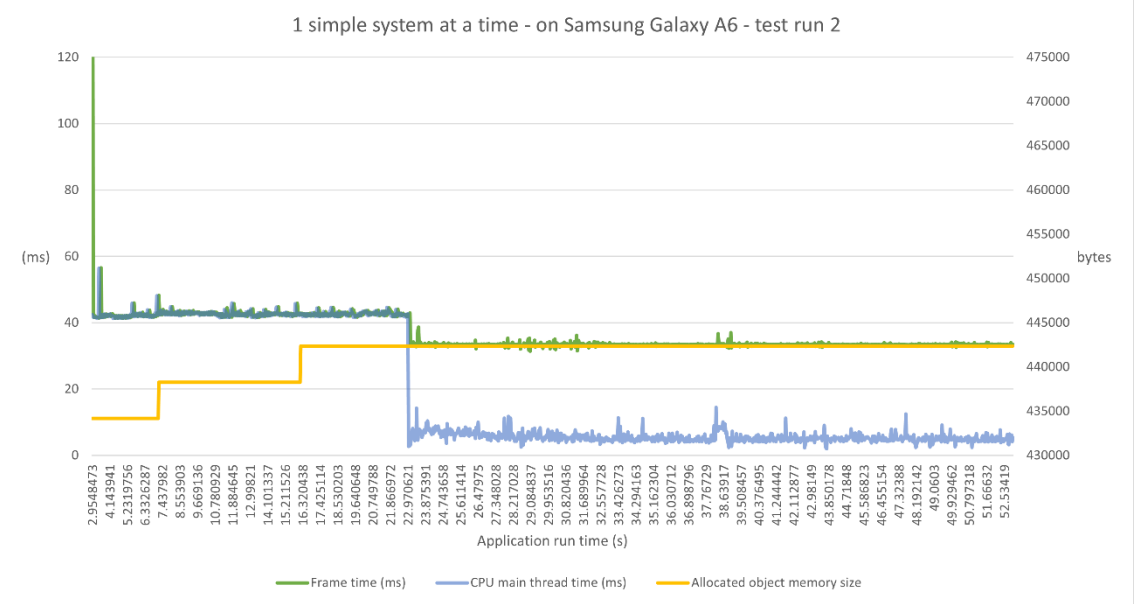

Figure 8. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, without pooling, simple systems.

If we compare the frame durations and memory use to equivalent application that uses Unity's ObjectPool<T0> class, we see similar frame durations and objects being created at the first few seconds. More memory is reserved early, which then reaches a similar amount as with the equivalent application that did not pool ParticleSystem objects. (Figure 9; Figure 10)

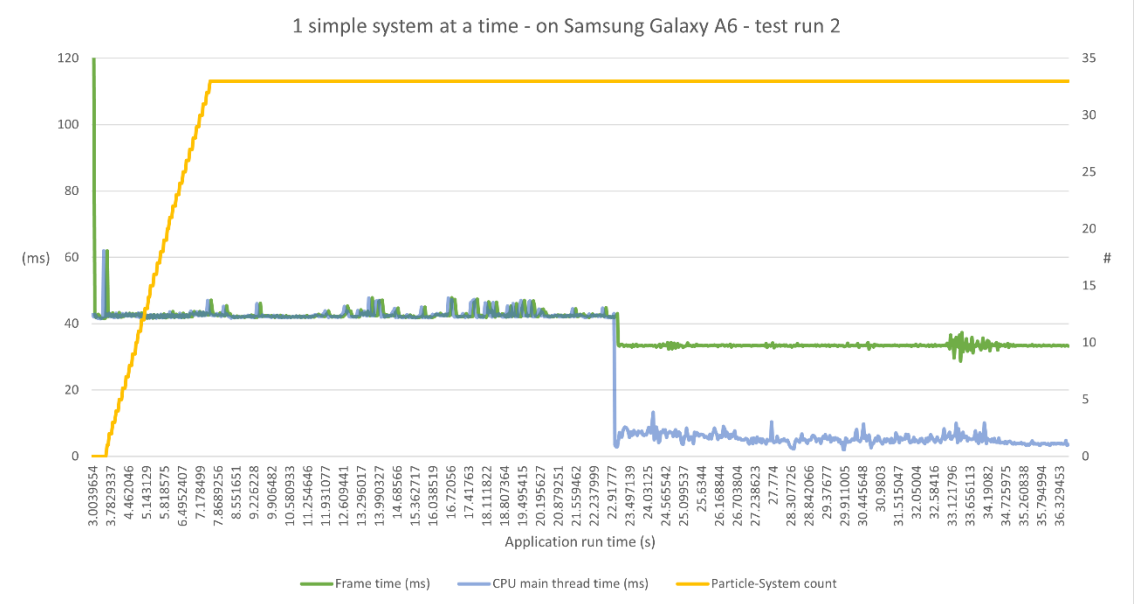

Figure 9. ParticleSystem count over 30 approximately seconds, with pooling, simple systems.

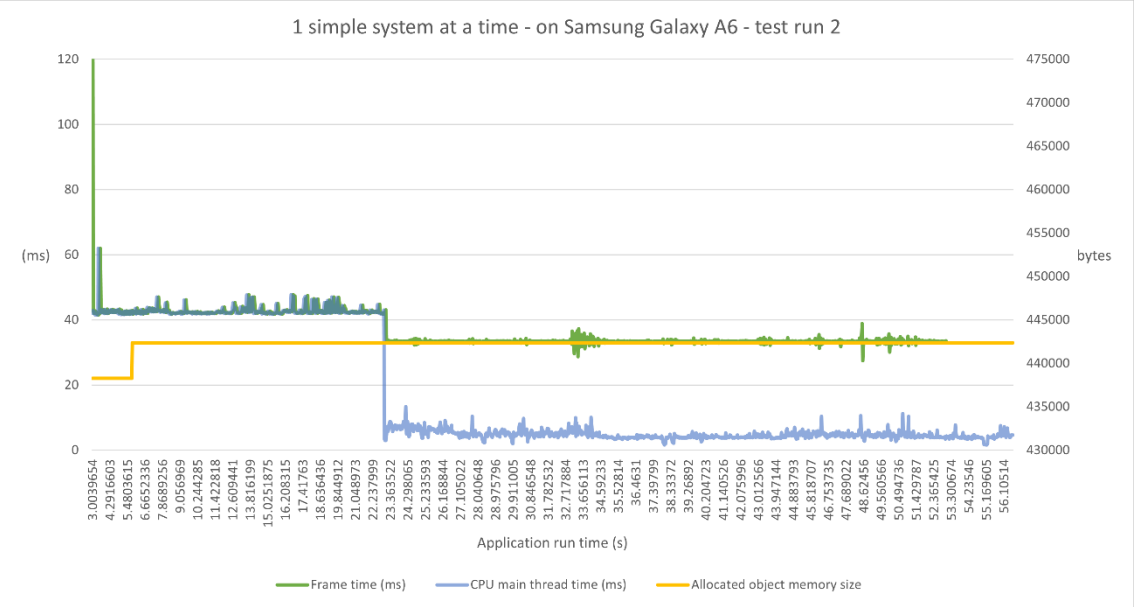

Figure 10. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, with pooling, simple systems.

However, if these applications are left to run for longer, the one that uses pooling is more efficient in its use of memory.

It should be noted that this is probably a very niche situation and that most games probably do not need to use independent ParticleSystems on the screen at a rate of 10 per second for a total of 100 seconds or more.

Notice that memory reserved for objects is greater than it was when application was set to create new ParticleSystems for only 20 seconds, and that this memory seems to not be released for a significant time after ParticleSystems are destroyed (Figure 11; Figure 12)

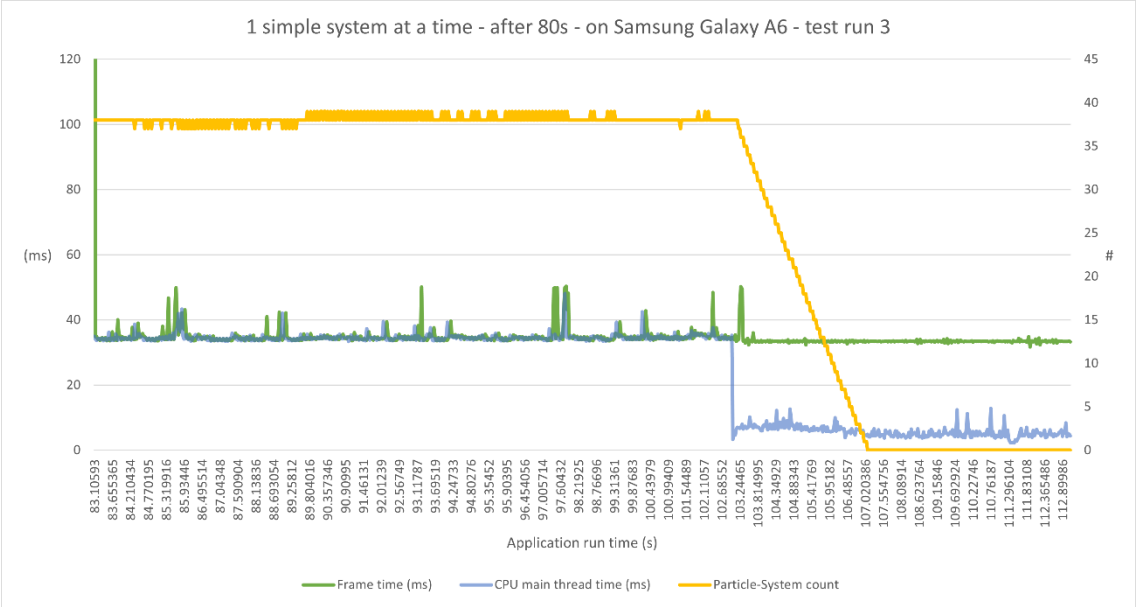

Figure 11. ParticleSystem count over 30 approximately seconds, after 80 seconds of run time, without pooling, on Samsung Galaxy A6

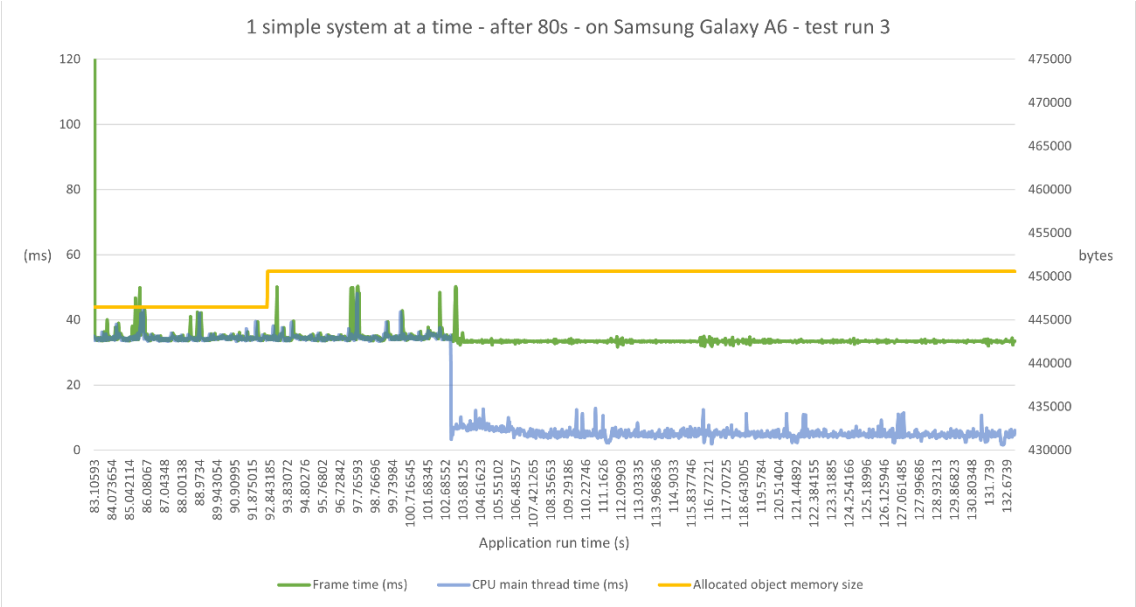

Figure 12. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, without pooling, simple systems.

In comparison, an equivalent application that used pooling predictably did not require noticeably more memory even over a total of 100 seconds of run time (Figure 13; Figure 14).

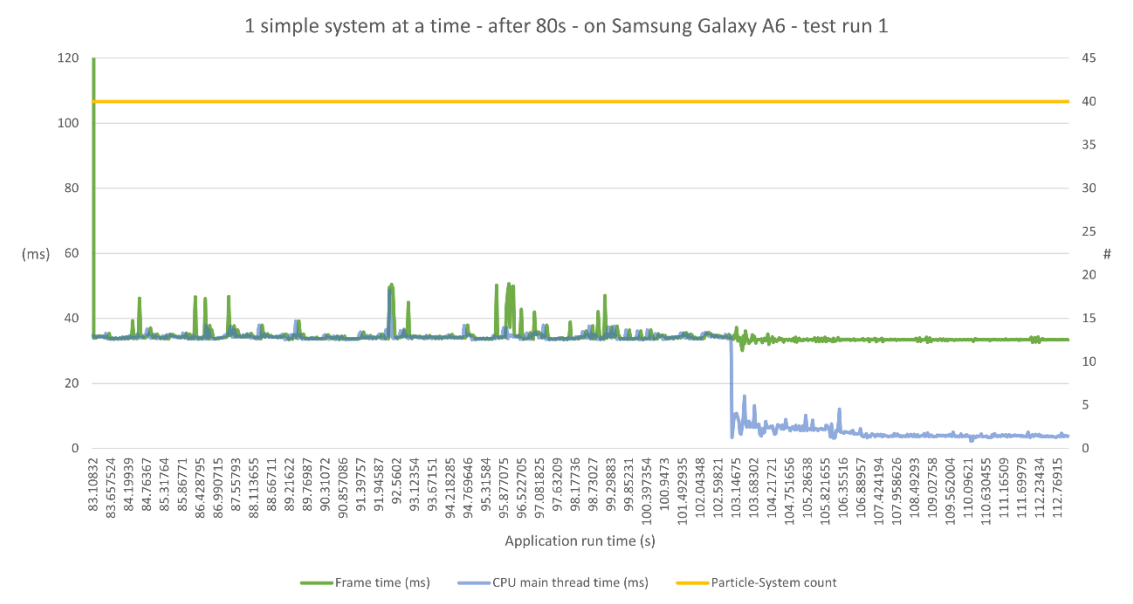

Figure 13. ParticleSystem count over 30 approximately seconds, after 80 seconds of run time, with pooling, simple systems.

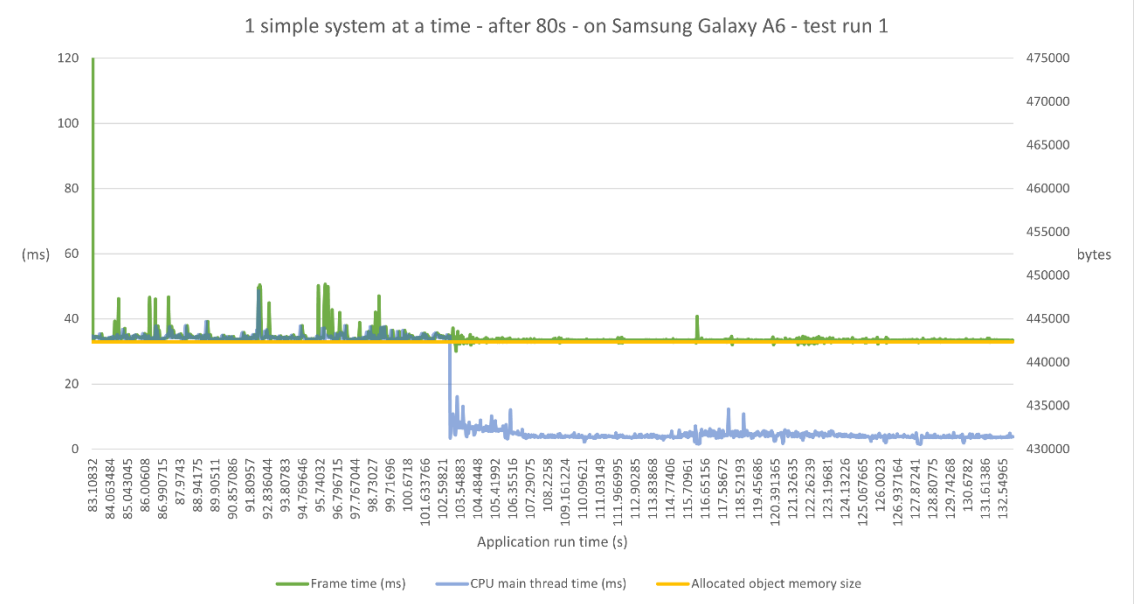

Figure 14. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, with pooling, simple systems.

Notice that decrease in main thread work times, depicted by the blue line, correlates well with the end of ParticleSystem use. The remaining charts presented will focus on memory, while keeping time measurement lines for context.

6.2 Rapid use of more complex ParticleSystems

ParticleSystems can have complex behaviors, such as particle size, color and / or velocity changes over time (Unity Technologies 2022f).

These were presumed to increase how much memory they use and presumably affect how long it would take for the CPU to create them. Following charts demonstrate how pooling can affect applications if they create more complex ParticleSystems instead of very simple ones (Figure 15).

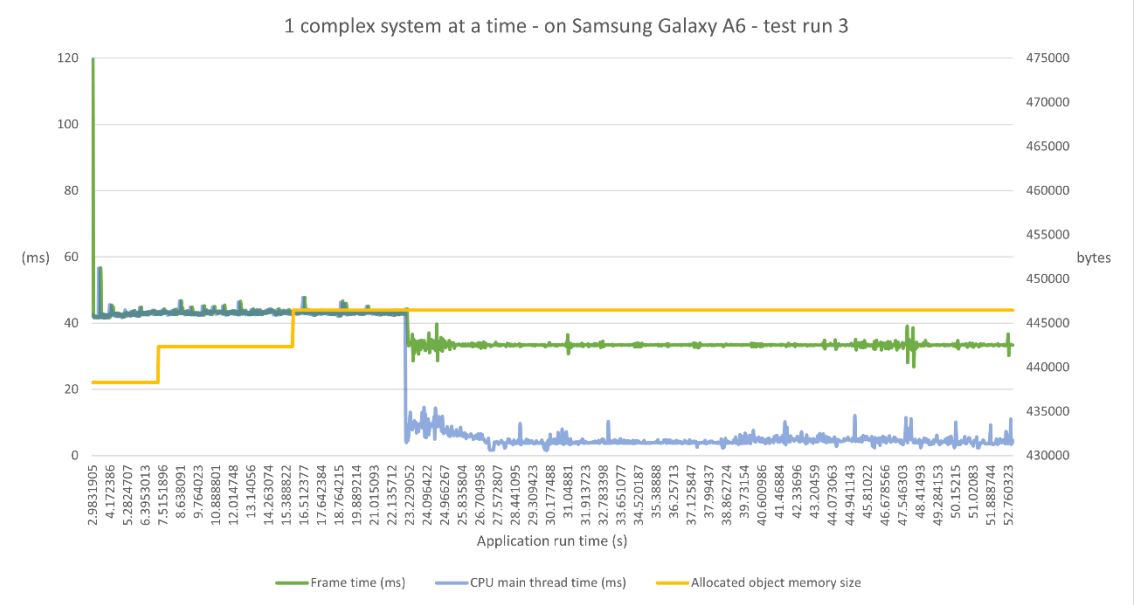

Figure 15. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, without pooling, complex systems.

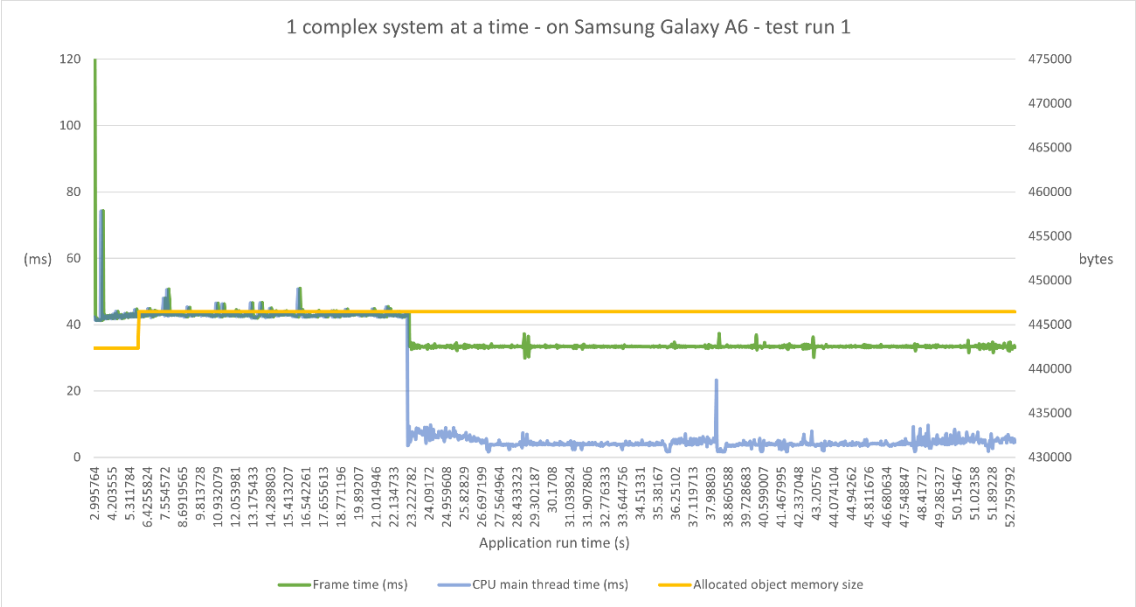

Figure 16. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, with pooling, complex systems.

The pattern remains the same, with both methods taking similar amounts of memory with rapid ParticleSystem use over 20 seconds. Like the applications that used simple ParticleSystems, results after a longer run time show that pooling can be more efficient in terms of memory use (Figure 17; Figure 18).

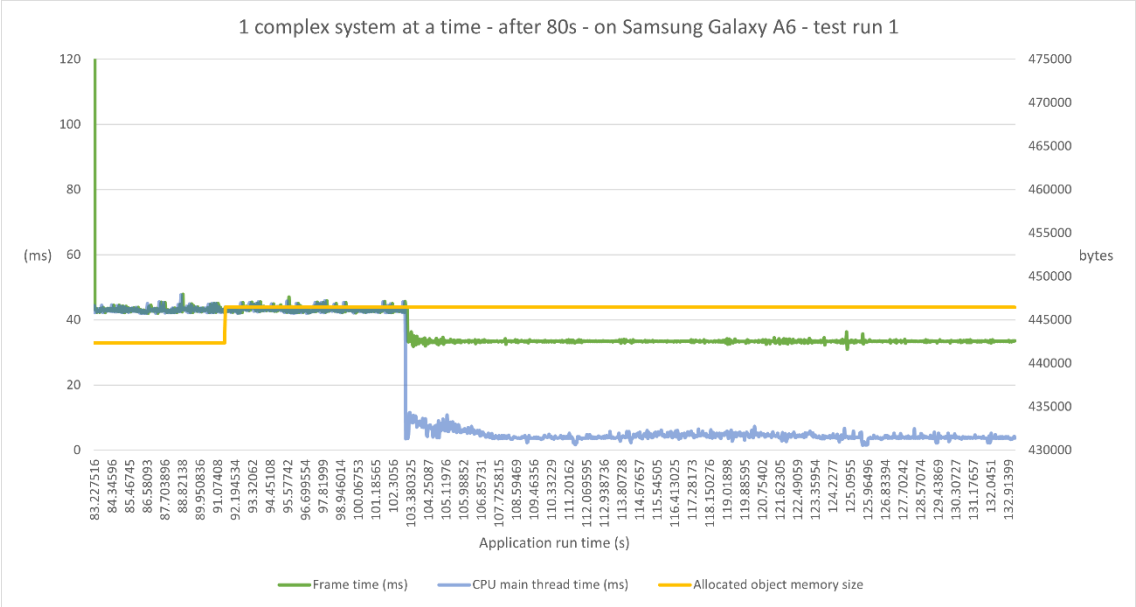

Figure 17. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, without pooling, complex systems.

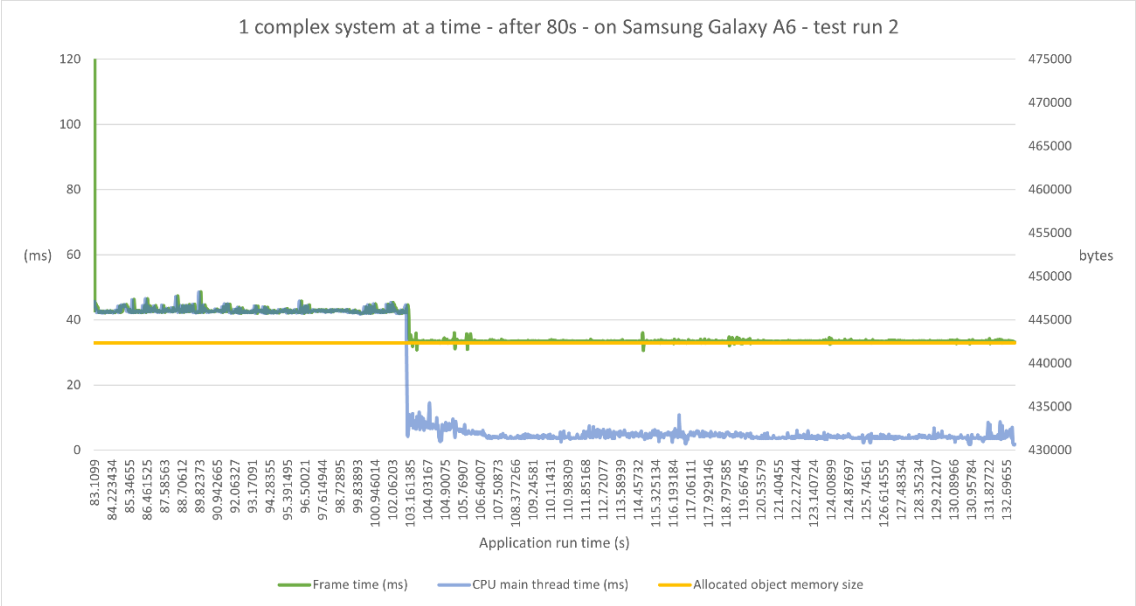

Figure 18. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, with pooling, complex systems.

Differences in memory use during short duration of ParticleSystem use became more pronounced when the quantity of ParticleSystems was increased to 3 at a time (Figure 19; Figure 20).

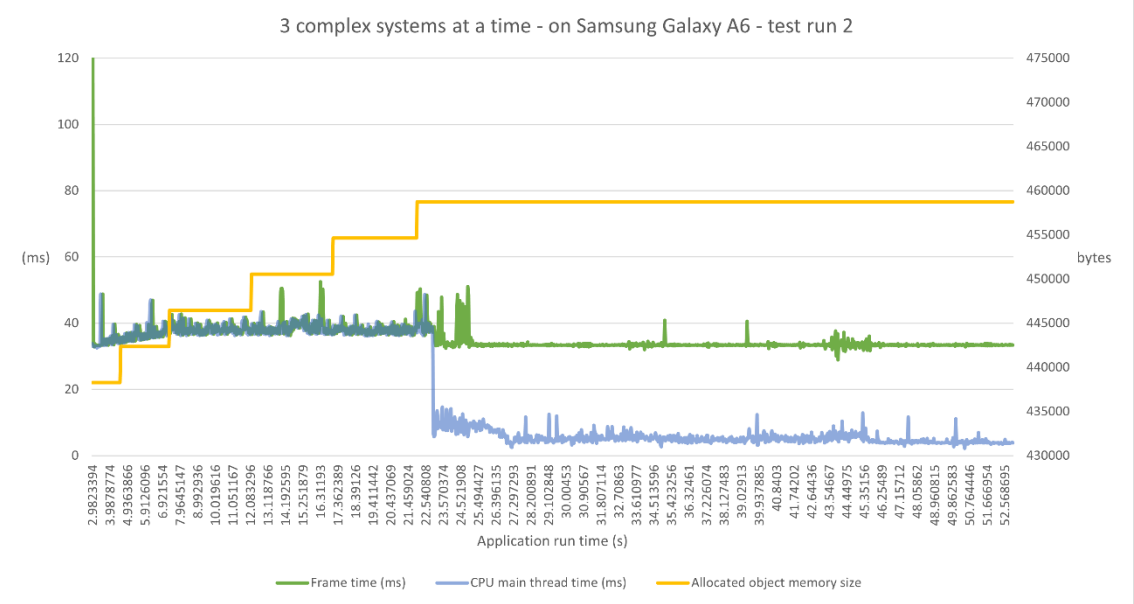

Figure 19. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, without pooling, 3 complex systems at a time.

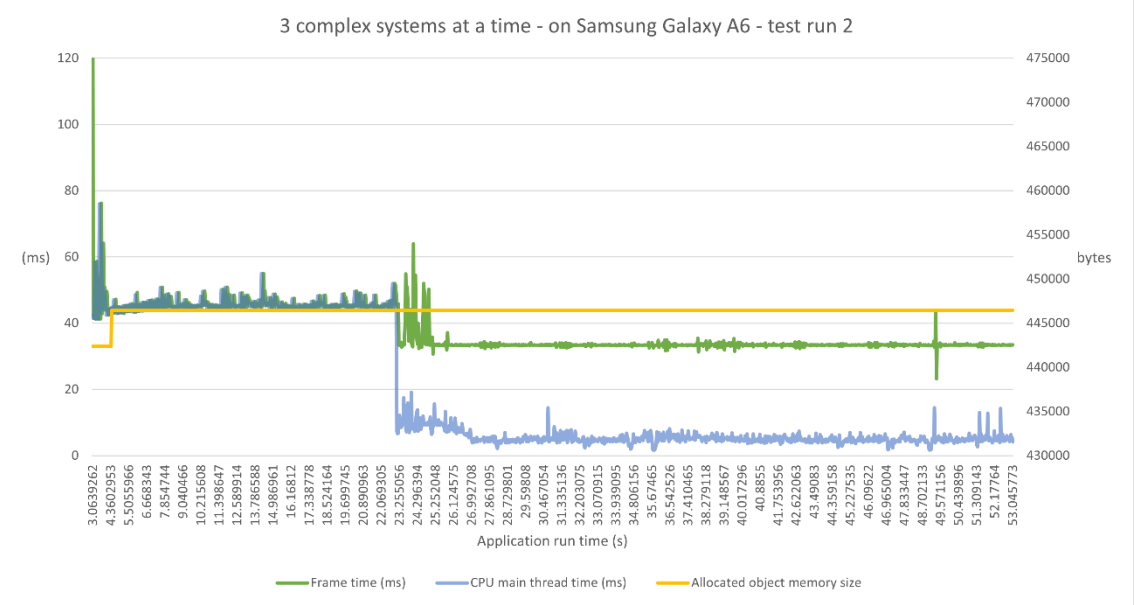

Figure 20. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, with pooling, 3 complex systems at a time.

6.3 ParticleSystem pooling with additional allocations in the background

It should be noted that these situations are not very realistic, as the applications have no complex behaviors that games could have. Notably, there are almost no other memory reserving mechanics in addition to ParticleSystems and arrays for recording profiling data. The following tests were done with same rapid rate of using ParticleSystems, but with additional, larger memory allocations at regular intervals.

Table 3. Additional memory allocation sizes and intervals

	Memory allocation size (bytes)	**Allocation intervals (seconds)**
Allocator 1	25600	1
Allocator 2	51200	1.5
Allocator 3	102400	4

With these additional allocations, differences in memory use amounts become less significant, although with even higher frequency of using ParticleSystems, such as 30 per second, more specifically 3 every 0.1 seconds, pooling can make a difference (Figures 25; Figure 26).

Figures 21 and 22 show memory reserved for objects fluctuating significantly over time with no object pooling being used (Figure 21; Figure 22).

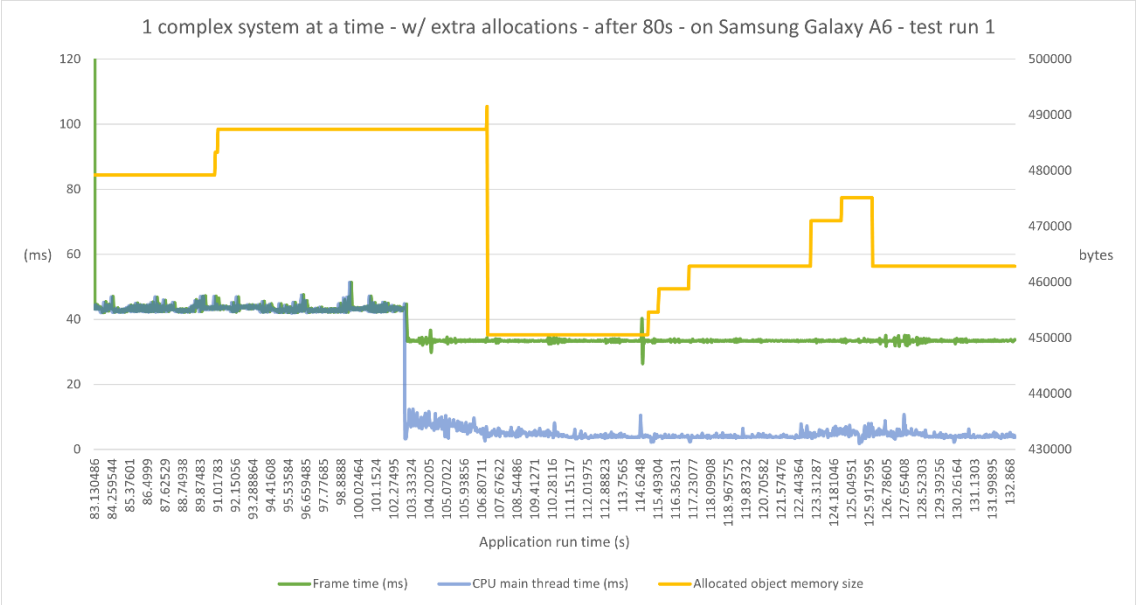

Figure 21. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, without pooling, complex systems, with additional allocations, 1.

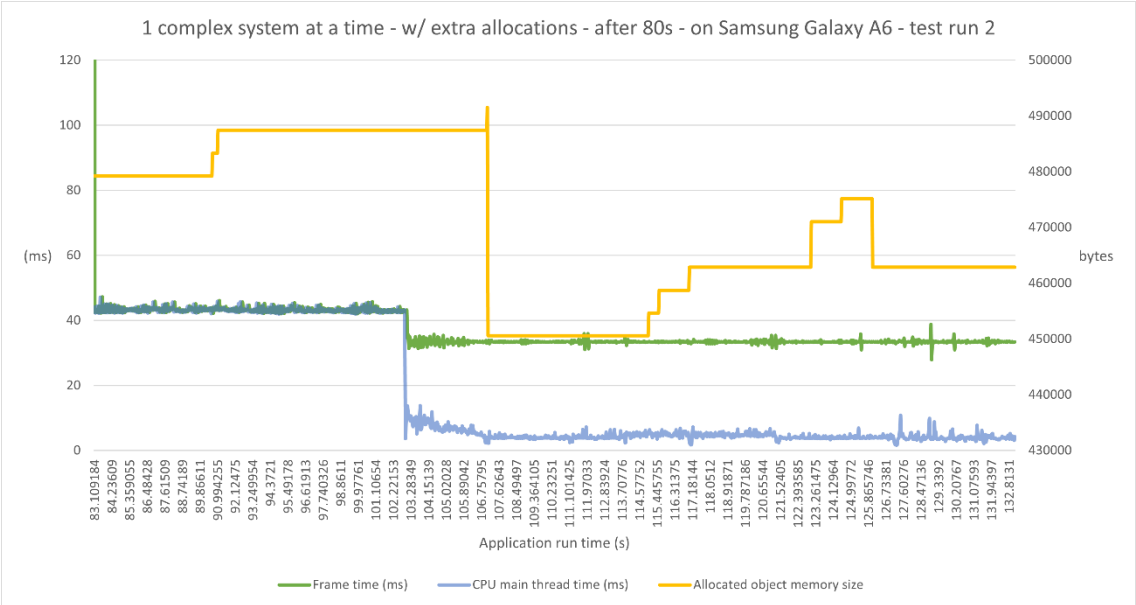

Figure 22. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, without pooling, complex systems, with additional allocations, 2.

With pooling, memory use follows a similar pattern, although the drop in the amount happens later in both test cases, and lowest amount of memory used for objects is higher (Figure 23: Figure 24).

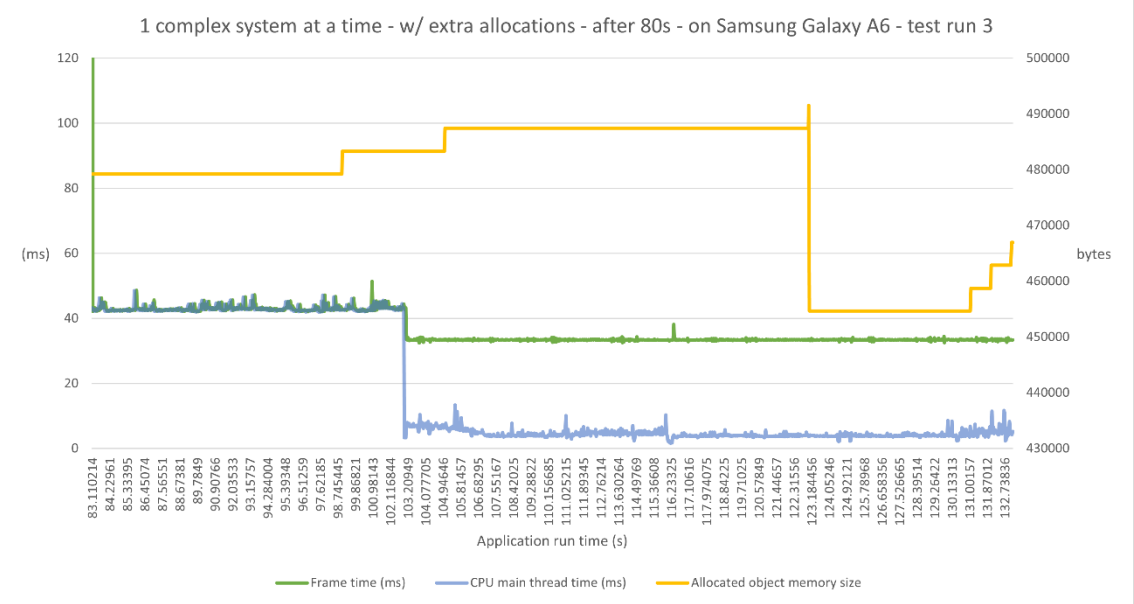

Figure 23. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, with pooling, complex systems, with additional allocations 1.

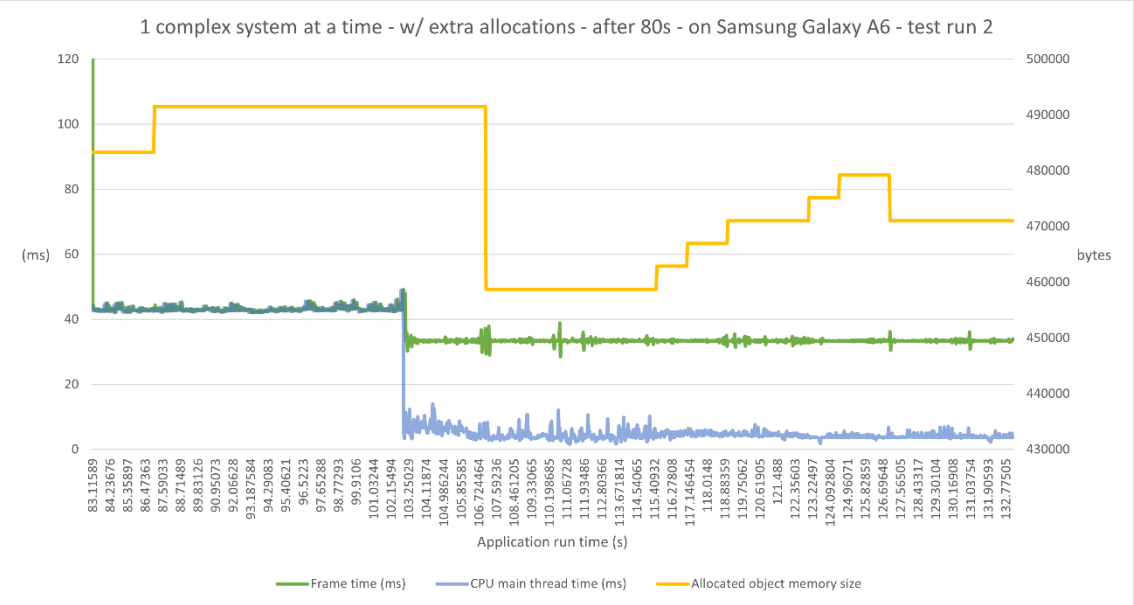

Figure 24. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, with pooling, complex systems, with additional allocations 2.

As previously mentioned, pooling can result in noticeable difference with excessive use of ParticleSystems with other significant allocations (Figures 25; Figure 26).

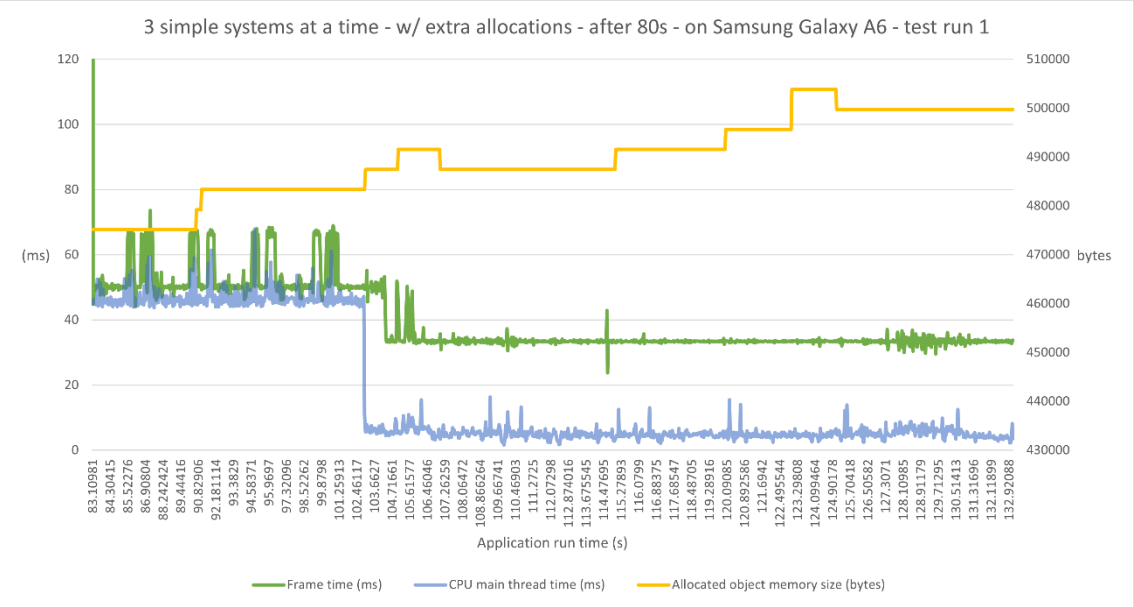

Figure 25. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, without pooling, 3 simple systems at a time, with additional allocations.

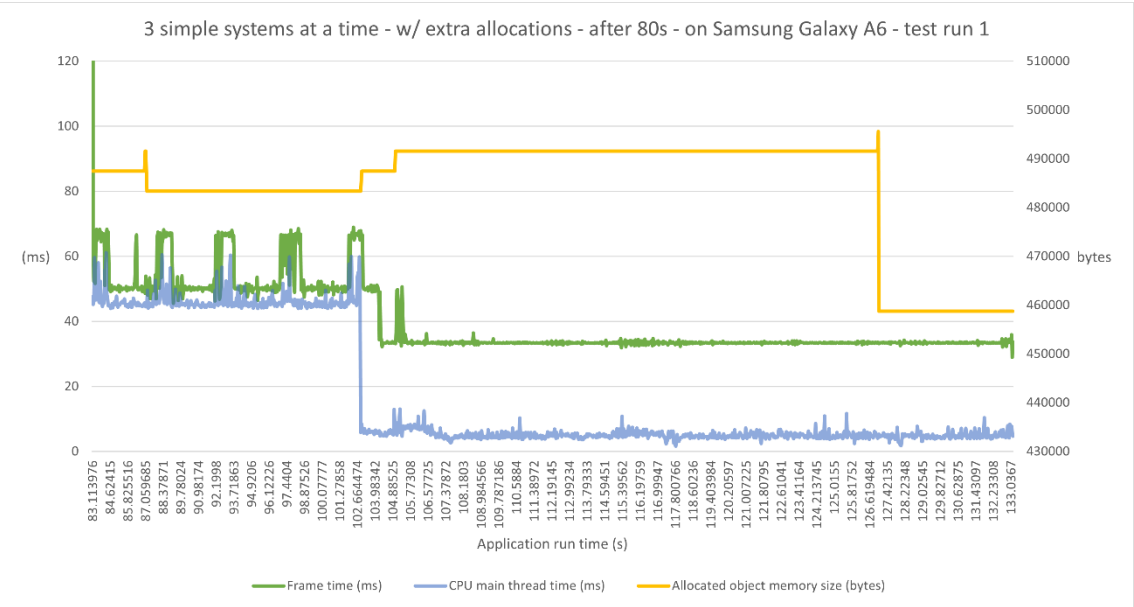

Figure 26. Memory used by live objects and memory not yet released by GC over approximately 50 seconds, after 80 seconds of run time, with pooling, 3 simple systems at a time, with additional allocations.

7 Conclusion

The purpose of this study was to compare the performance between Unity applications that use Instantiate method to create new ParticleSystem objects and applications that use object pool pattern, specifically Unity's ObjectPool<T0> class.

Testing of excessively large object pools was not successful due to the implementation of Unity's ObjectPool<T0> class.

Different implementations of object pools may be worth testing. Unity's ObjectPool<T0> class was used because it comes together with Unity's other features and allows for growth of pool size if needed, but there may be more efficient implementations.

With just ParticleSystems being pooled or created regularly, there was a noticeable difference in the amount of memory used. However, these situations are not necessarily realistic, as many games would have other objects created and destroyed during their run-time. With additional memory allocations at regular intervals, the differences can become less noticeable compared to memory use overall, unless the frequency at which particle effects are needed is excessively high.

On the other hand, the additional allocations are something that should probably be avoided and may be a better focus for object pooling. It may be that ideally almost all often-used objects should be pooled to achieve the best performance.

ParticleSystems that do not use additional behaviour modules, such as particle size change over time, seem to be easy and cost effective to create, which means that unless the GameObjects that ParticleSystems are attached to also have components that require more work from the CPU to create, implementing an object pool for them probably should not be a priority during development. On the other hand, performance costs of ParticleSystem pooling also seem to

be small, and if implementing an object pool for them is not too difficult in a game, it may be worth implementing.

The tests raised several questions regarding testing performance itself, as there are numerous factors that can affect results and the software design of Unity applications / games can vary in structure.

Measured total times between frames and main thread work times at each frame are close to each other when ParticleSystem object are actively created or requested from the pool, but recorded values for main thread work times are oocasionally higher than total times between frames, which should not be possible. This makes the reliability of these measurements questionable. Times between frames were derived from the measured time from the application start at each frame, which suggests that either Unity's clock was inaccurate or that main thread work times are inaccurate, or possibly both.

The names of the produced log files could be more descriptive, to make processing them easier, and the binary to CSV converter should probably be able handle entire folders with subfolders to make accessing test data easier.

More importantly, recording data points over time produces a lot of charts, while showing variation in data, making analysis of the gathered data a slow process

More definitive results may be possible to obtain with different analysis and profiling tools, such as Memory Profiler component of Android Profiler and Unity Profiler. The decision to not use the latter was made due to concern of Unity Profiler taking up too much of device resources, but this may not be an issue.

References

Albahari, B. Albahari, J. 2015a. ‘Chapter 2. C# Language Basics’ in C# 6.0 in a Nutshell : The Definitive Reference. 6th ed. ProQuest Ebook Central: O'Reilly Media. pp. 42. ISBN 978-1-491-92706-9 (printed).

Albahari, B. Albahari, J. 2015b. ‘Chapter 14. Concurrency and Asynchrony’ in C# 6.0 in a Nutshell : The Definitive Reference. 6th ed. ProQuest Ebook Central: O'Reilly Media. pp. 564. ISBN 978-1-491-92706-9 (printed).

Boehm, H-J. Demers, A. J. 2014. A garbage collector for C and C++. Referenced 7.6.2023. https://www.hboehm.info/gc/

Boehm, H-J. Dubois, P. F. 1995. Dynamic Memory Allocation And Garbage Collection. American Institute of Physics. Referenced 1.6.2023. https://pubs.aip.org/aip/cip/article-abstract/9/3/297/509106/Dynamic-Memory-Allocation-and-Garbage-Collection?redirectedFrom=fulltext

Bonet, R. T. 2021. Object Pooling in Unity 2021+. The GameDev Guru. Referenced 1.6.2023. https://thegamedev.guru/unity-cpu-performance/object-pooling/

Chen, J. Guo, R. 2020. Stack and Heap Memory. CS 225. Referenced 1.6.2023. https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/

Christou, T. Ioannis. Efremidis, S. 2018. To Pool or Not To Pool? Revisiting an Old Pattern. arXiv. Referenced 1.6.2023. https://arxiv.org/abs/1801.03763

Digital Preservation Home. 2021. CSV, Comma Separated Values. Format Description Categories. Referenced 1.6.2023. https://www.loc.gov/preservation/digital/formats/fdd/fdd000323.shtml

Duffy, D. J. Germani, A. 2013. ’24.4 An Introduction To Threads In C#’ in C# for Financial Markets. pp. 638. ISBN: 978-1-118-50281-5 (electronic). 978-0-470-03008-0 (printed).

Finto. 2018. OS/2. AFO - Natural resource and environment ontolog. Referenced 1.6.2023. https://finto.fi/afo/en/page/?uri=http%3A%2F%2Fwww.yso.fi%2Fonto%2Fyso%2Fp20555

Goetz, B. 2005. Java theory and practice: Urban performance legends, revisited. developerWorks. Referenced 1.6.2023. https://web.archive.org/web/20111229023158/http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html

Google. 2023a. Android Debug Bridge (adb). Android Developers. Referenced 1.6.2023. https://developer.android.com/tools/adb

Google. 2023b. Android Runtime (ART) and Dalvik. Android Developers. Referenced 1.6.2023. https://source.android.com/docs/core/runtime

Google. 2023c. Referenced 1.6.2023. Enable Developer options. Android Developers. https://developer.android.com/studio/debug/dev-options#enable

Google. 2023d. Inspect your app's memory usage with Memory Profiler. Android Developers. Referenced 1.6.2023. https://developer.android.com/studio/profile/memory-profiler

Google. 2023e. Manage your app’s memory. Android Developers. Referenced 1.6.2023. https://developer.android.com/topic/performance/memory

Google. 2023f. Memory allocation among processes. Android Developers. Referenced 1.6.2023. https://developer.android.com/topic/performance/memory-management

Google. 2023g. Overview of memory management. Android Developers. Referenced 1.6.2023. https://developer.android.com/topic/performance/memory-overview

GSMARENA. 2023a. Samsung Galaxy A6. Referenced 1.6.2023. https://www.gsmarena.com/samsung_galaxy_a6_(2018)-9155.php

GSMARENA. 2023b. Samsung Galaxy J5. Referenced 1.6.2023. https://www.gsmarena.com/samsung_galaxy_j5-7184.php

Ionin, K. 2023. ParticleSystemPoolingTestingProject. GiHub. Referenced 8.6.2023. https://github.com/jodeConstant/ParticleSystemPoolingTestingProject

Koulaxidis, G. Xinogalos, S. 2022. Improving Mobile Game Performance with Basic Optimization Techniques in Unity. MDPI. Referenced 1.6.2023. https://www.mdpi.com/2673-3951/3/2/14

Lemonaki, D. 2021. What is CPU? Meaning, Definition, and What CPU Stands For. freeCodeCamp. Referenced 1.6.2023. https://www.freecodecamp.org/news/what-is-cpu-meaning-definition-and-what-cpu-stands-for/

Microsoft. 2021. Garbage collection. Documentation. Referenced 1.6.2023. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/

Microsoft. 2023a. BinaryWriter Class. Documentation. Referenced 1.6.2023. https://learn.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.8

Microsoft. 2023b. Fundamentals of garbage collection. Documentation. Referenced 1.6.2023. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

Microsoft. 2023c. Memory allocation - Generations. Documentation. Referenced 1.6.2023. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals#generations

Microsoft. 2023d. The managed heap. Documentation. Referenced 1.6.2023. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals#the-managed-heap

Nystrom, R. 2021. Object Pool. Game Programming Patterns. Referenced 1.6.2023. http://gameprogrammingpatterns.com/object-pool.html

Oxford University Press. 2023. Definition of RAM noun from the Oxford Advanced American Dictionary. Oxford Learner's Dictionaries. Referenced 1.6.2023. https://www.oxfordlearnersdictionaries.com/definition/american_english/ram_2

SourceMaking. 2023. Object Pool Design Pattern. sourcemaking.com. Referenced 1.6.2023. https://sourcemaking.com/design_patterns/object_pool

Unity Technologies. 2022a. GameObject. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/ScriptReference/GameObject.html

Unity Technologies. 2022b. GetMonoUsedSizeLong. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/2022.1/Documentation/ScriptReference/Profiling.Profiler.GetMonoUsedSizeLong.html

Unity Technologies. 2022c. Incremental Garbage Collection. Unity Documentation. Referenced 7.6.2023. https://docs.unity3d.com/Manual/performance-incremental-garbage-collection.html

Unity Technologies. 2022d. Instantiate. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/2022.1/Documentation/ScriptReference/Object.Instantiate.html

Unity Technologies. 2022e. MonoBehaviour. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Unity Technologies. 2022f. ParticleSystem. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/2022.1/Documentation/Manual/class-ParticleSystem.html

Unity Technologies. 2022g. Prefabs. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/2022.1/Documentation/ScriptReference/ParticleSystem.html

Unity Technologies. 2023a. Debugging on an Android device. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/Manual/android-debugging-on-an-android-device.html

Unity Technologies. 2023b. FrameTiming. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/2022.1/Documentation/ScriptReference/FrameTiming.html

Unity Technologies. 2023c. FrameTimingManager. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/2022.1/Documentation/Manual/frame-timing-manager.html

Unity Technologies. 2023d. Garbage collection best practices. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/Manual/performance-garbage-collection-best-practices.html

Unity Technologies. 2023e. Garbage collector overview. Unity Documentation. Referenced 1.6.2023. https://docs.unity.cn/Manual/performance-garbage-collector.html

Unity Technologies. 2023f. Managed memory. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/Manual/performance-managed-memory.html

Unity Technologies. 2023g. ObjectPool<T0>. Unity Documentation. Referenced 1.6.2023. https://docs.unity3d.com/ScriptReference/Pool.ObjectPool_1.html

Wienholt, N. 2004. 'Garbage Collection and Object Lifetime Management.' in Maximizing .NET Performance. Apress, Berkeley, CA. pp. 101–119. ISBN: 978-1-4302-0784-9 (electronic). ISBN: 978-1-59059-141-3 (printed). DOI: 10.1007/978-1-4302-0784-9_7