MALMÖ UNIVERSITET

Teknik och samhälle
Datavetenskap och medieteknik

Examensarbete

15 högskolepoäng, grundnivå

Vilken effekt har olika lumpsamlare på bildhastigheten i en spelmiljö

What Effect Does Different Garbage Collectors Have on the Framerate of a Game Environment

Hans Oskar Emanuelsson Östlund

Examen: kandidatexamen 180 hp
Huvudområde: datavetenskap
Program: Spelutveckling
Datum för slutseminarium: 2018-05-24

Handledare: Magnus Krampell
Examinator: Åse Jevinger

Sammanfattning

I detta arbetet undersöks vilka olika effekter en lumpsamlare har på en spelmiljö. Detta mäts genom att räkna hur många objekt existerar i spelmiljön och hur hög bildhastighet spelmiljön arbetar på. För att testa detta så designades ett enkelt shoot-’em-up spel. I spelet styr en primitiv AI ett rymdskepp som skjuter på en grupp asteroider som datorn skapar. Problemet för spelet är att datorn konstant skapar dessa asteroider och skeppet skapar kulor för att skjuta på dem. Det finns inte en övre gräns hur många asteroider och kulor som kan finnas i spelet. Detta problemet blir värre för att datorn tar inte bort dessa objekt. Under spelets gång uppdateras båda typerna av objekt. Detta resulterar i att den ökande mängden med objekt kommer dra ner hur hög bildhastighet spelet arbetar på.

För att testa hur lumpsamlare påverkar spelet implementerades tre enkla lumpsamlare. I detta fallet är det “Reference counting”, “Mark-Sweep”, och “Mark-Compact” lumpsamlare. För att jämföra med hur dessa påverkade spelet så gjordes också en version som tog bort objekten manuellt Versionerna med lumpsamlare och versionen som själv tog bort objekt förväntades att kunna spela spelet med en bildhastighet på 45 FPS. 45 FPS är en hastighet som inte påverkar en spelares prestation medans lägre hastigheter kan negativt påverka en spelares prestation.

Testerna visar att versionerna med lumpsamlare presterade sämre än versionerna som tog bort objekten manuellt. Det var enbart den manuella versionen som kunde agera på över 45 FPS och den versionen hade också lägsta antalet objekt i spelet. Testen var dock orättvist designad mot lumpsamlare versionerna då de var tvungna att arbeta med ett ansträngande test för en dator. Detta var menat men det är fortfarande mer extremt än vad som normalt är förväntat att en lumpsamlare ska behöva arbeta med. Ett annat problem som bara berörde Mark-Compact lumpsamlaren var att den stora fördelen med en Mark-Compact Lumpsamlare är inte utnyttjad i detta testet. Denna fördel är att under normala förutsättningar kan det bli “hål” i ett minne när information tas bort, Mark-Compact lumpsamlaren sparar över dessa hål med information.

Abstract

This thesis explore the effects that garbage collection has in a real time game. This is measured by counting the number of objects and the framerate in a game that is intentionally created to be very taxing memory-wise. A game was designed in a simple shoot-’em-up style. A simple AI is in control of a spaceship shooting at constantly spawning asteroids. Neither the asteroids nor the bullets will be deleted when the program is running. This results in a growing number of objects which will slow down the framerate noticeably. Both types of objects will continue to be updated, thus becoming a larger and larger burden for the game to handle.

To test how garbage collectors would affect the game three garbage collection algorithms were implemented in the game, in this case; a Reference Counting Collector, a Mark-Sweep Collector, a Mark-Compact Collector, and one version of the game that manually deletes objects. Each garbage collector and the manually handled game was expected to be able to run above 45 Frames per second while having the lowest number of object in play simultaneously. 45 FPS is a framerate that does not efet how well a player can play a game. Lower framerate on the other hand can have a negative effect on a players performance.

The conclusion of this paper is that the versions of the game with the garbage collectors performed worse than the manual deletion. Only manual deletion was able to keep above the 45 FPS requirement and had a lower number of objects in play. The test was however unfairly stacked against the garbage collectors as the test forced them to have a heavy workload. Another problem that was unique for the Mark Compact Garbage Collector is that the main issue that a Mark Compact Garbage Collector addresses is not accounted for in the program. The advantage a Mark Compact Garbage collector provides is that normal garbage collectors can create “holes” in the memory when information is deleted. Mark Compact Garbage Collectors solve this issue by overriding these holes with information when cleaning up the memory.

Terminology

Dynamic Memory Allocation - Memory allocation done at run-time, rather than at initialization of the program.
Fps - Abbreviation of frames-per-second, the unit of measuring frame rate.
Frame rate - The rate at which the system updates the game-state.
Garbage - Memory occupied by objects that are no longer in use by the program.
Garbage Collection - A form of automated memory management, where garbage is reclaimed by a Garbage Collector. Abbreviation: GC
Stall - when the program execution is halted in order to shift or free memory.

Table of Contents

1 Introduction **1**
1.1 Background 1
1.2 Research Questions 2

2 Technical Description **3**
2.1 Garbage Collectors 3
2.2 Three types of Garbage Collectors 4
2.2.1 Mark-Sweep 4

2.2.2 Reference 4
2.2.3 Mark-Compact 4
2.3 The effect of Framerate on Gameplay 5
2.4 List of Requirements 5

3 Related Work **7**
3.1 Dynamic Memory Allocation/Deallocation Behavior in Java Programs 7
3.2 Perspectives, Frame Rates and Resolutions: It's all in the Game 7
3.3 Nonblocking Real-Time Garbage Collection 8

4 Method **9**

4.1 Method description 9
4.2 Game description 10

5 Results **12**
5.1 Introduction 12
5.2 Garbage Collection Algorithms 12
5.2.1 Mark Sweep Garbage Collector 13
5.2.2 Reference Counting Garbage Collector 14
5.2.3 Mark-Compact Garbage Collector 15
5.2.4 No Garbage Collector 16

6 Analysis **17**

7 Conclusion **19**

References **20**

Appendix **23**

1 Introduction

1.1 Background

According to Ravenbrook Limited's memory management glossary; "Garbage collection (GC), also known as manual memory management, is the manual recycling of dynamically allocated memory. Garbage collection is performed by a garbage collector which recycles memory that it can prove will never be used again." (Ravenbrook Limited, 2016). The Garbage collector achieves this memory recycling by looping through the objects in a program while the normal operations of the program is temporarily halted. The objects that the garbage collector can prove as not in use are erased to free up memory. The primary use of Garbage Collectors is to free up memory that is either inaccessible by the program or memory that is wasted on objects that are not in use. Doing this manually often requires doing "tedious and error-prone" work, which according to Ritzau is the reason "developers find [Garbage Collectors] attractive" (Ritzau, 2003). The alternative to garbage collection would be to use Dynamic Memory Allocation and Deallocation techniques, which require the programmer to specify exactly which memory gets allocated/deallocated and when. Using GC instead of Dynamic Memory techniques would allow developers to prioritize their time in other areas of development.

The idea behind this thesis is based on a tweet of Daniel Benmergui, programmer of the game "Ernesto". Benmergui claimed in a tweet that "in order to optimize [his] game", he would have to "go out of [his] way to prevent the Garbage Collector from doing anything" (Daniel Benmergui, 2015). He is not the only one that believes Garbage Collectors can have a negative impact on performance. These views are also shared by Matthew Hertz and Emery D. Berger from the university of Massachusetts. They claim that a garbage collector may (when compared to Dynamic Memory Allocation) require "five times as much memory" in order to "[match] or exceed (by up to 9%) the runtime performance of the best explicit memory manager" (M. Hertz & E.D.Berger, 2005). The software and hardware developer Apple Inc. also cited these drawbacks as the reasons for not including a garbage collector in their operating system in 2003 saying "Unfortunately garbage collection has a suboptimal impact on performance" (Apple inc, 2011). However this is not to say that Garbage collectors are not useful. Thomas Padron-McCarthy argues that it quickly becomes problematic for a developer to know what can be thrown away and when (Padron-McCarthy, 2008).

The fact is that garbage collectors can, in such a time-sensitive program as a real time computer game, cause noticeable breaks. If a Garbage Collector is added to a game the gameplay will freeze temporarily when the garbage collector puts the code on hold to clean it up. However this might not be universal for games, as games that are turn based might not be as affected. This issue is intriguing because it raises questions if garbage collectors, for all their good contributions, are even worth having in a game. The exact parameters of how much a garbage collector can slow down a game. Which garbage collector is able to slow down the program the least while keeping a game running for long periods of time is an issue that needs to be addressed and examined. This lead to the reason why this topic is chosen as the subject of research in this thesis. There are strong opinions on the topic among programmers both for and against garbage collectors, and it is more impactful to see exactly how this would affect a game with concrete numbers.

1.2 Research Questions

To find out what these concrete numbers are and to determine exactly what tests are needed are precise and consistent research questions a requirement. To facilitate this does this thesis revolve around two questions. This is a number that was chosen to get neither too narrow nor too vague results. At the same time must the questions be simple to test and provide clear and presentable results from which one can draw conclusions. The research questions were for all these reasons formulated as follows;

RQ1: Which out of any of these garbage collectors and a game that manually removes objects itself will be able to run a game at a framerate above 45 Frames per seconds.

RQ2: Which of these garbage collectors or a game that manually delete objects will be able to run with the least amount of objects in play.

Obviously, the framerate can be improved by choosing a more powerful computer, but in this case, the results are compared to a reference implementation, where “garbage collection” is done manually (by the programmer). This reference implementation is able to provide a framerate of 47 fps on average (i.e. above the 45 fps limit) and has an average of 17 objects in game.

2 Technical Description

2.1 Garbage Collectors

A garbage collector is a process that manually deletes memory not in use by a program. This has the advantage of manually cleaning up memory that the program can then reuse. This is then performed by a subsystem that is formally termed as the garbage collector (Padron-McCarthy, 2008). Garbage Collectors can be divided into two categories: Stop-the-world and Incremental. According to Martin Thompson Stop-the-world Garbage Collectors work by completely halting a program's execution while freeing memory (these occurrences are referred to as "Stalls" in this thesis) (M.Thompson, 2013). Paul Thomas describes incremental garbage collection as the process of collecting garbage in several smaller chunks, thus resulting in shorter (but more frequent) stalls (P.Thomas, 2010).

However, not all garbage collectors are equal when it comes to memory management. One issue that arises with certain types of garbage collectors is when an object is deleted from a list, there will be a vacant spot in the memory. This vacancy will be only accessible very inefficiently due to what is called "the fragmentation problem". For instance a garbage collector would clear out a gap in the memory that is eight bytes wide. If the program needs to save a new piece of information that is nine bytes large it would not be able to save this new information in the eight bytes gap but instead it needs to search for a bigger memory space to save the information in.

One can see a representation of the Fragmentation Problem in the below table(see: Table 1). At first three bytes are used to save information. On the second row the second byte is deleted. This leaves an empty void at the third row. At the fourth row two new bytes worth of information are added, but these two bytes cannot fit in the second spot they are instead saved on the 4th and 5th positions. At the fifth row the potential problem that this creates is noticeable. Five positions are used to save all of our information, the second position however is not used to save any information at the moment. While this mere one byte is not currently an issue, the gaps continue to grow. This can lead to a situation where huge chunks of data is only wasted space causing programs to use more memory than they need to.

1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

Table 1: Representation of the empty byte of memory that the fragmentation memory would leave.

2.2 Three types of Garbage Collectors

2.2.1 Mark-Sweep

A mark sweep garbage collector works in two steps: Mark and Sweep. At first it goes through the mark step. In this step, the garbage collector loops through a list of objects that are checked one by one. If an object in the lists not referenced to by other object it is unable to be accessed by the program anymore. The Mark Sweep Garbage collector will then flag it as marked by a boolean value attached to each object. Once this step has been completed, the Garbage Collector goes on to the sweep step. Once again it goes through all objects with the same loop approach as in the mark step. This time however it only checks if an object is marked or not. The marked objects are then deleted by the garbage collector. The second step in this process is to help the garbage collector with cyclic structures with reverse referencing and allow it to run easier because it is not forced to maintain a count on every object references as a reference counter would be forced to(K.Fox, 2014).

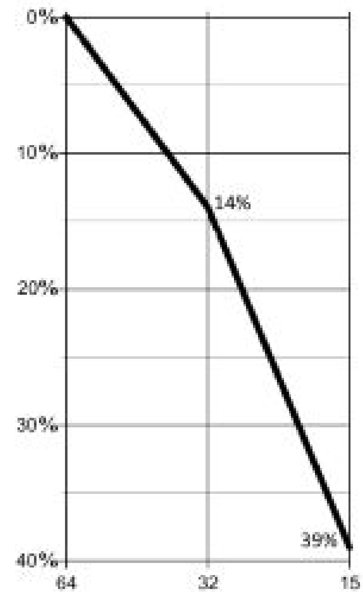
2.2.2 Reference Counter

A reference counting garbage collector is a garbage collector that keeps track of how many references there are to an object. To simplify this the references was changed to an int value that is either one or zero in the game designed for this test. The reasoning behind this is that the game is simplistic enough not to require references and the int value serves as an appropriate simulation. If an object reaches zero in this value it is considered referenceless. The garbage collector starts by looping through the objects.The Garbage Collector deletes any object that has zero references to it. If for an example an object is referenced to by the deleted object would the new object have one less reference to it. This could bring the total number of references down to zero and the new object would be deleted as well. The reference counting algorithm has issues since a cyclic structure with reverse references is something that it cannot sort through. Apple once released a reference counting garbage collector that did not properly work with their operating system and they had to replace it with a mark sweep garbage collector instead (K.Fox, 2014).

2.2.3 Mark-Compact

The mark compact garbage collector is similar in function to the mark sweep garbage collector. It first marks all objects in the same manner as the mark sweep garbage collector. The difference is in the sweep/compact phase where instead of just deleting objects which would leave gaps in the memory of unused bytes the mark compact “moves” the memory back. Filling in the memory gaps with the old memory before saving new data at the end (B. R. Preiss, 1998). This Garbage collector has the advantage of rather than deleting objects and moving on, the new empty memory slot is filled with old memory, and the new information saved at the end. This helps to solve the fragmentation problem that was described in chapter 2.1. Out of the three collectors it is the only one that addresses this issue (K.Fox, 2014).

2.3 The effect of Framerate on Gameplay



The framerate of a game should to be at least 45 frames per second when the garbage collector is included because it is considered optimal for games (B. F. Janzen and R.J. Teather, 2014). The framerate is especially important in a game, as proven in the study made by B. F. Janzen and R.J. Teather in which they measured the framerate's effect on gameplay by having a player press squares that appear on the screen and measuring the time between the presses. The players effectiveness at completing the task given to them by the game decreases by 14% when a game runs at 32fps compared to 64fps and with 39% when 15fps is compared to 64fps (B. F. Janzen and R.J. Teather, 2014). Mark and Kajal Claypool describes the difference in framerate in a shooting game as having a "Significant" impact on the gameplay and the player (M. Claypool and K. Claypool, 2009). The need for the game to run at high frames per second without long interruption means that a garbage collector does not have infinite time during which it can run. Instead the garbage collector must pause its loops if it has surpassed its allotted time. This means that a garbage collector could potentially be tasked with an continuously growing backlog that will over time slow down the game below the desired framerate.

2.4 List of Requirements

To ensure that Garbage Collectors are working properly and not causing noticeable stalls must certain requirements be put on them. According to Martin Schoeberl and Wolfgang Puffitisch a garbage collector must fulfill two requirements (M.Schoeberl and W.Puffitisch, 2010).

<u>Requirement 1 - Length of Stall</u>

Requirement 1.1: the first requirement is to "ensure that programs with bounded allocation rates do not run out of memory". This is achieved by deleting objects that would not be used by the program.

Requirement 1.2, which is to "provide short blocking times" which is also the second requirement the games needs to achieve. The stalls (called "blocking time" by Schoeberl and Puffitisch) may not be longer than 100ms. This requirement is related to the requirement 1.2 as laid out by Schoeberl and Puffisch in their text where they called it "to provide short blocking times".(M.Schoeberl and W.Puffitisch, 2010) This 100ms delay is too short for a player to notice that there is a stall according to Miller. According to him only a "Skilled keyboardist" would be able to notice this when typing (Miller, R.B, 1968).

The short delay will also be quick enough for the inputs made by the player to not feel slow or unresponsive. When it comes to digital games, players expect – just as with any other digital interface –

that any action performed should have an immediate reaction (Miller, R. B, 1968), else they may begin to question if the game is functioning properly. Whenever the user performs an action, the program should respond accordingly. The shorter the time, the more responsive the system feels. In order for something to be experienced as being instantaneous, it must take less than 100ms (100 milliseconds, or 0.1 seconds) (Miller, R.B, 1968).

Requirement 2 - Framerate

The framerate for a program that uses a garbage collector needs to be at the lowest 45 FPS when the garbage collector is running because it is considered optimal for games according to B.F Janzen and R.J. Teather. Any lower than 45FPS and the gameplay experience would begin to worsen and the player effectiveness begins to suffer. This is due to the slow frame rate affects a players capacity to react to a moving target negatively (B. F. Janzen and R.J. Teather, 2014). As this is the absolute minimum for a game to be acceptable, this framerate was chosen as a threshold for this evaluation

3 Related Work

3.1 Dynamic Memory Allocation/Deallocation Behavior in Java Programs

A similar issue to the one about garbage collectors is tackled by Anthony S. Fong in his paper where he observes that games developed in Java are not very well optimised (A.S.Fong 2002). However Anthony Fong looked at the effect of Dynamic memory allocation rather than Garbage collection. The problem that he describes with the memory allocation is evidenced by the fact that “Even a simple Othello applet game easily requires half a million memory allocations for only one game play“. While Anthony Fong does not describe how the situation could be improved, he claims that the number of memory allocations is “extremely large”. The game developed by Anthony Fong used as many as 650,468 allocations during one session. These allocations varied greatly in size. Some were simple one byte allocations but others reached 64,128 bytes in size.

This issue was tested by developing a program that intentionally caused poor performance. Anthony Fong arrived to the conclusion that in gameplay, poor handling of memory is a risk that could cause severe performance issues (A.S.Fong 2002). Anthony Fong even suggested that memory was “one of the most significant causes of slow execution in Java programs“.

3.2 Perspectives, Frame Rates and Resolutions: It’s all in the Game

The problem that low framerate can have on a players performance is explored by Claypool, M. and Claypool, K. The Claypools who tested their findings with 27 human test subjects and measured the test subject’s performance in a game that was developed to have 6 different stages, two with an isometric, two with a first person, and two with a third person perspective (M.Claypool & K.Claypool 2009). These stages were tested with different framerates and resolutions to test the effect of resolution and framerates on the game. The conclusion reached was that framerate in games directly links to how quickly a player will react. Resolution on the other hand was deemed to be less influential as there were no noticeable difference between 800x600 pixels and 1280x1024 pixels which is the difference between a game shown on a low end computer and a high end one respectively.

3.3 Nonblocking Real-Time Garbage Collection

Another thesis that also evaluates a garbage collector in a real time situation,was one written by Martin Schoeberl and Wolfgang Puffitsch (M.Schoeberl and W.Puffitsch 2010). It evaluates the garbage collector found in what the authors call “java processors”, however this is limited to the java processors rather than a more general study of garbage collectors. To measure the effectiveness of the Java processors M.Schoeberl and W.Puffitsch overload the garbage collector, This was achieved by opening and running several applications simultaneously but it is not clear from M.Schoeberl and W.Puffitsch’s paper how the results were measured.

They do however offer a way that could make a game easier for garbage collectors such as only copying lists of objects with an interruptible copy that can be interrupted by a garbage collector when running (M.Schoeberl and W.Puffitsch 2010). This is achieved by having the forward pointer that evaluates what object to copy stored in another area compared to the copy function. Interestingly enough, they also tested the program without garbage collectors, and in this case the program ran with the lowest irregularities in time between each run, suggesting a more efficient and less memory draining way of handling a program.

4 Method

4.1 Method Description

The method is a controlled Experiment, this is a test that will change one or a few variables while the rest of the variables are kept constant through the test. In the case of the experiment described in this paper, the garbage collectors will be the changing variables while the game itself will be the constant one. (S.Easterbrook, J.Singer, M.A.Storey, and D.Damian 2008). The environment is a game that goes through four treatments, where each treatment is a new version of the game with different variables.

1. a game manually handling the memory drains
2. a game running with a Mark-sweep garbage collector
3. a game running with a reference counter garbage collector
4. a game running with a Mark-compact garbage collector

The Controlled experiment was chosen because of its simplicity as well as being able to provide reliable and stable results with few random elements.

The first step of this controlled experiment is to create a game to test the garbage collectors. It needed to be capable of creating large numbers of objects rapidly. This is done in order to increase memory usage. Quality of the game's design *as a game* is irrelevant. It only needs to be able to create the objects and have a situation where the objects will be deleted. Thus, a game of relatively simple design is designed: an arcade-style "shoot-em-up" game similar to "Space Invaders" and "Galaxian". Unlike those games, the one created for this thesis is very taxing memory-wise in order to test the effectiveness of the different garbage collection algorithms. The relative simplicity of the game's design also allows for more time to be spent on the effectiveness and testing of the various garbage collectors rather than the game itself.

Step two is to run the four different treatments, as mentioned above one treatment will be the game that manually handles the memory drain. The three other treatments uses garbage collectors to handle the memory drain. The game measured the framerate and the number of objects in play and saves this information in a txt document before it starts its garbage collector. To get the total numbers of objects the number of asteroids and bullets are simply added together. For the treatment that manually handles the memory drain the treatment saves this information when the other treatments runs their garbage collectors. The garbage collectors is required to run for only 0.1 seconds while keeping a framerate above 45.

Step three is to analyse the results the game collected. The framerate and the number of objects in play at the same time is compared with the results of the other treatments to see how they affect the framerate as well as the number of objects compared to one another. The objects are the cause of the memory drain in the game. The less objects there are the higher frames per second the game is able to run at.

4.2 Game description

The game controls a spaceship object that create bullet objects. The asteroid objects are created at random intervals during the course of the game. These two objects (asteroids and bullets) are the main issues in the program that will need to be cleaned up. All objects are in the games's memory for as long as the program runs, as they are never explicitly destroyed unless a garbage collector or the manual versions delete them. Any objects that are off-screen will effectively be considered garbage by the garbage collectors when they loop through the objects.When there is too much garbage in play the game will begin to slow down due to all the calculations it needs to make to keep all objects moving.

The game works by the ship moving a constant speed on the x axis. This speed is set to either 5 or -5 pixels. A pixel is the smallest single component of a digital image, in this case the game window and is represented as a single square of one colour.. When the ship has reached its end position (either position 0 or 600 on the x axis) its speed will change between 5 and - 5. Every 10th frame a new asteroid object will be created and added to a list of asteroid objects.

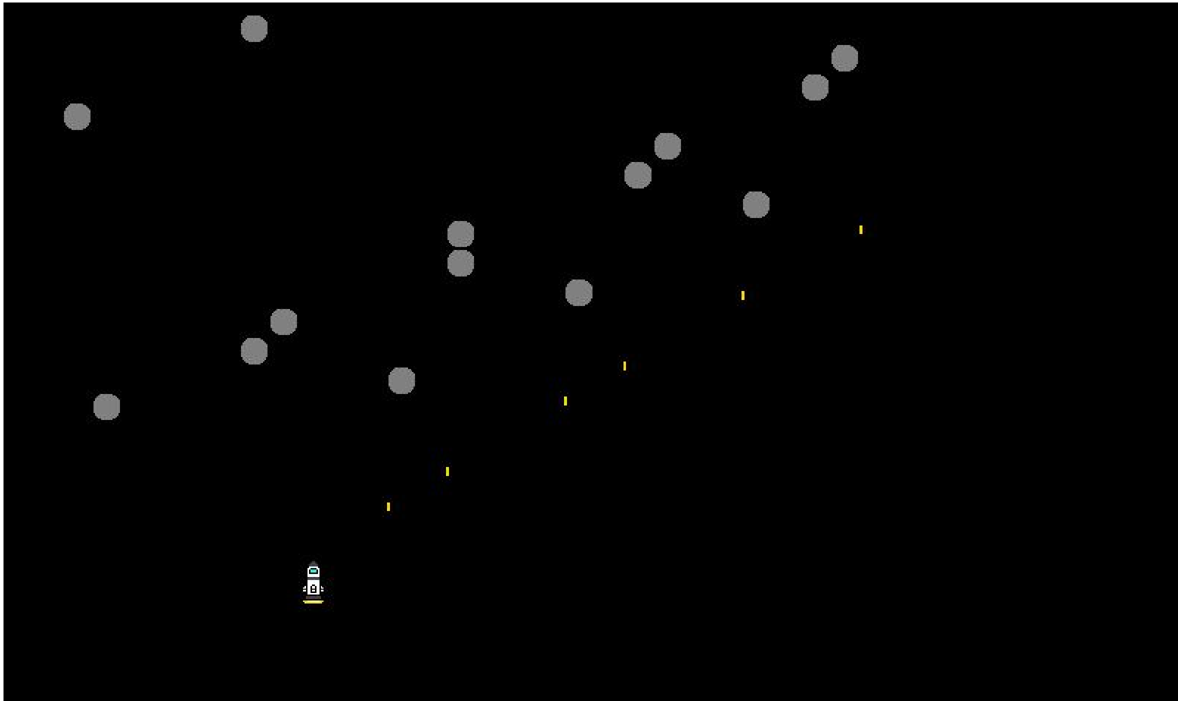


Fig2 Game in play.

The position asteroids are created at is determined by randomly generating a number between 0 and 30 and then multiplying it by 20. The asteroid will then move down the Y-axis by 2 pixels per frame. When the ship is on the same X-axis as an asteroid it will create a bullet. This is achieved by a new bullet object being placed in a list of bullet object. This bullet will move up on the y axle with a speed of 2 pixels per frame. When the game loops through the list of bullets it will also once again loop through the list of asteroids and see if the bullet intersects with an asteroid. When this occur they will both be taken out of the game and thus be considered removable by the garbage collectors according to the methodology.

A countdown is also in counting down the time until the garbage collectors will pause the game and clean it up.When the countdown hits zero the game will save the total number of objects, asteroids, bullets, and framerate. Once this has been saved, the garbage collector will start to clean up the objects. How this is achieved is determined by which garbage collector is running in the version of the game that is played and is determined before the game is started. When the garbage collector has run out of its allotted time which is 0.02 seconds, the game will again return to updating the objects. Once this update phase is complete the game will move on to the draw phase where the game once again loops through each list of bullets and asteroids, and the ship and draws each object on the screen.

If the memory is Allocated Dynamically, objects should be deleted by the game as soon as they are not used anymore. Referring specifically to the game designed for this thesis, the asteroid and bullet objects' memory could be freed as soon as they no longer need to be drawn on screen. While it can be not only advantageous but also necessary for a game to update objects out of view, there is no reason for the game to actually draw these objects. If this game were designed properly with the game itself manually deleting objects when appropriate would there not be a need for the game to pause and run a GC. This is exactly what the treatment that manually handles memory do.

5 Results

5.1 Introduction

The results in this section of the thesis are presented as graphs, the Y axis represents either the number of objects in play or the frames per second the game is running at depending on the graph. The X axis represent how many times the garbage collector has been activated by the game. Each graph contains the first 40 times the garbage collector is activated by the game and this number is chosen arbitrarily because any larger number made it difficult to read the graph. However, since the tests followed similar patterns not much data was lost after the arbitrarily chosen 40 first iterations. The average number of Frames per Second, or Objects is calculated by adding the total number of Frames per Second or Objects from the 40 values on each graph and then divide it with 40 to get an average result.

In the Graphs that shows the amount of objects in play before the garbage collector is activated three lines are included. These Lines are the Total number of Objects in play which is represented by the black line, the numbers of Asteroids in play which is represented by the grey line, and the number of Bullets in play which is represented by the blue line.

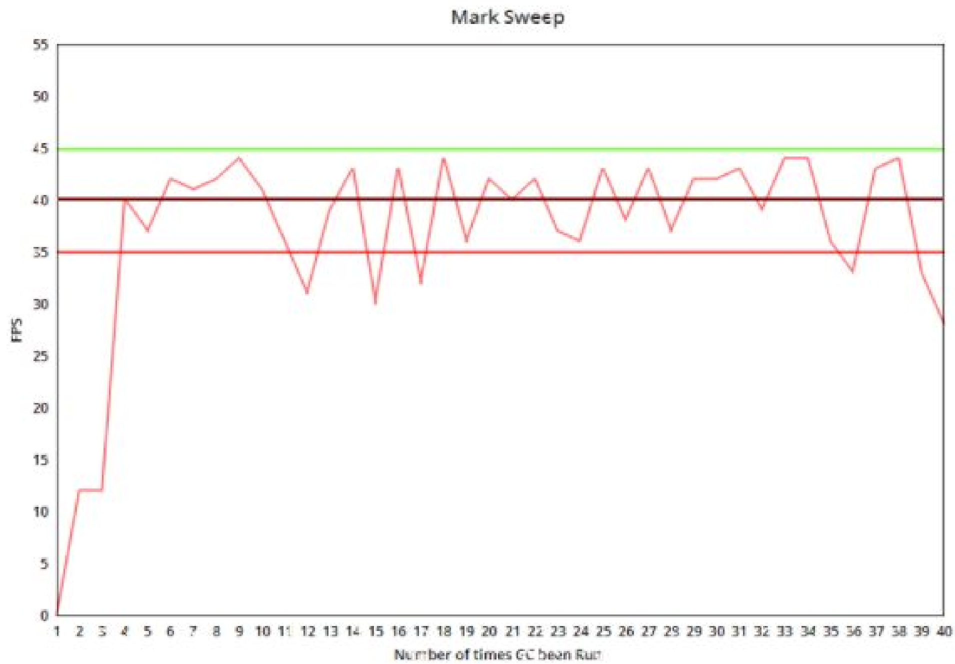
The Graphs representing the Frames per Seconds the game is running at has three lines besides the results on them. The red line represents the average Frames per Second the game is running at and the green line represents the required 45 Frames Per Second that the game needs to achieve. The Dark red line is the average of all frames per second minus the first five to compensate for the slow framerate in the start of the game. The average is calculated by adding the total amount frames per second and then dividing them by 40 in the case of the total average, and adding result 6 to 40 and dividing it with 35 in the case of the average without the first five iterations. With this in mind, it is easy to consider why each algorithm is selected for the test and how their unique approaches to the problem affect the number of objects in play and how high or low framerate they are able to hold.

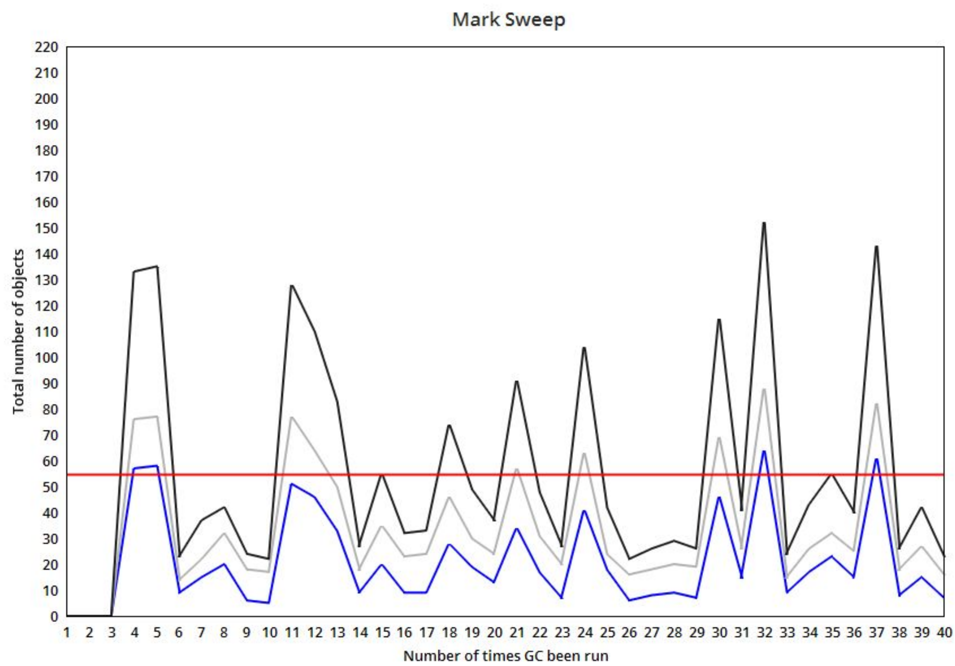
The results are in the following pages presented as two graphs, the first graph showing how high FPS the game is running at and the second graph will show the total number of objects in play. In the first graph does the green line represents the required 45 FPS the game needs to run at. The dark red line represents the average FPS the game was running at after the first five iterations and the light red line represents the average total FPS through all iterations. In the second graph the red line represents the average total number of objects through all 40 iterations of the Garbage collector.

5.2 Garbage Collection Algorithms

The Mark Sweep Garbage Collector is selected due to its simplistic nature and it being one of the more commonly used garbage collectors. The Reference counting Garbage Collector is similarly quite simplistic but has a different approach to GC than Mark-Sweep and Mark-Compact. Lastly, Mark-Compact Garbage collector is selected since it addresses the fragmentation problem caused by the two previous collectors. All of these are Incremental Garbage Collectors due to the game is played in real-time, thus emphasizing being able to play without noticeable interruptions.

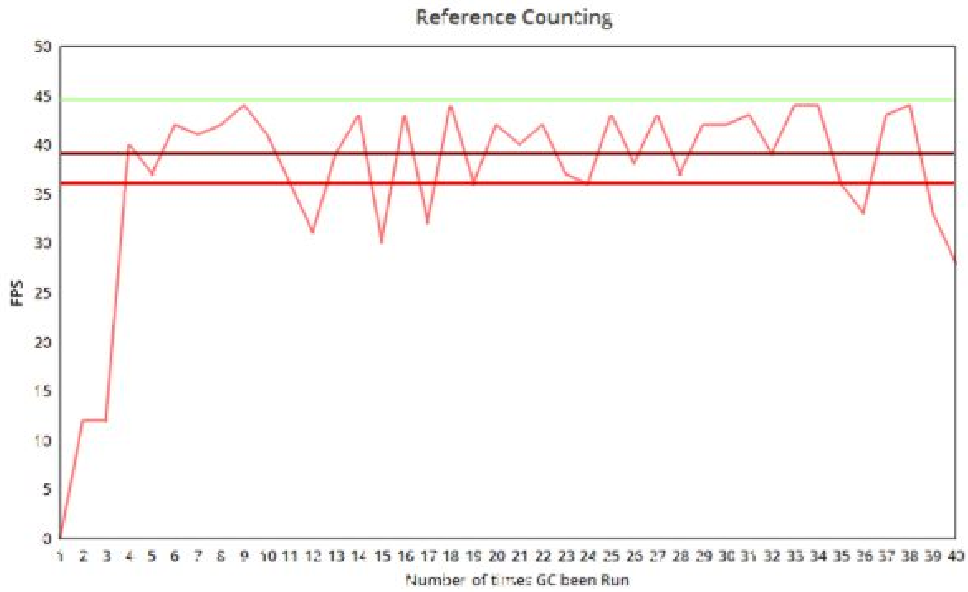
5.2.1 Mark Sweep Garbage Collector

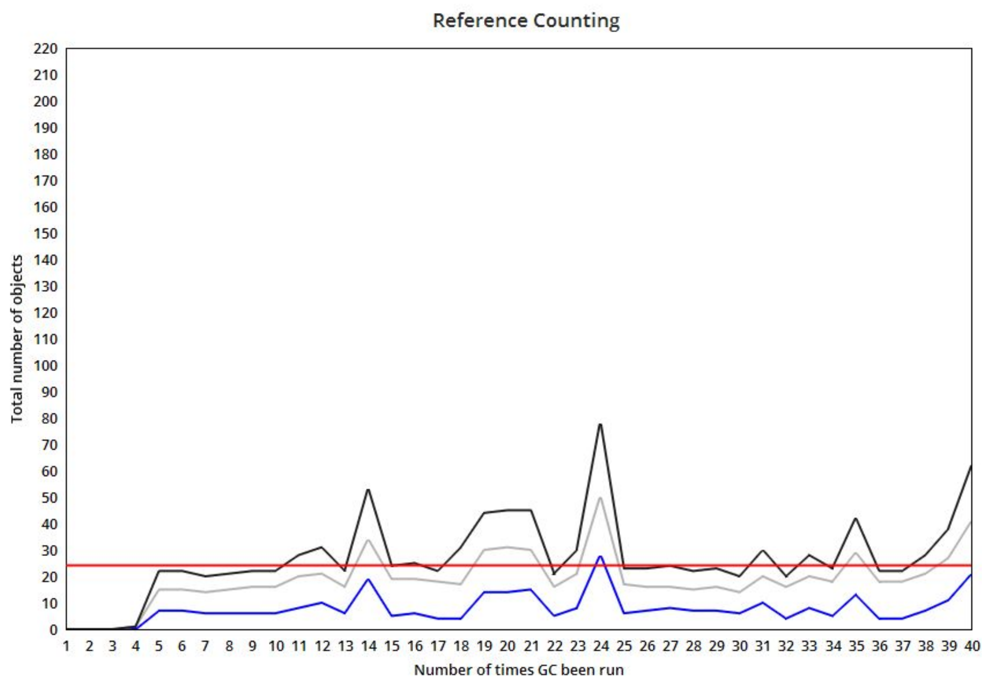




The Mark sweep garbage collector is running on an average of 35 FPS and at 40 as an average without the first five iterations which is below the required 45 FPS but it also is not able to reach 45 FPS once. The average number of objects in play is 54.

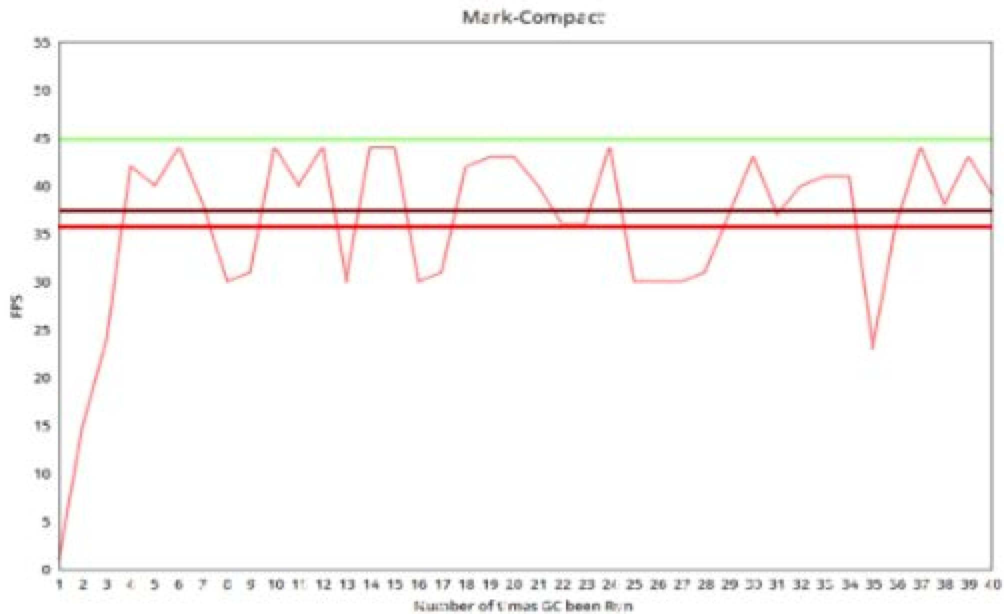
5.2.2 Reference Counting Garbage Collector

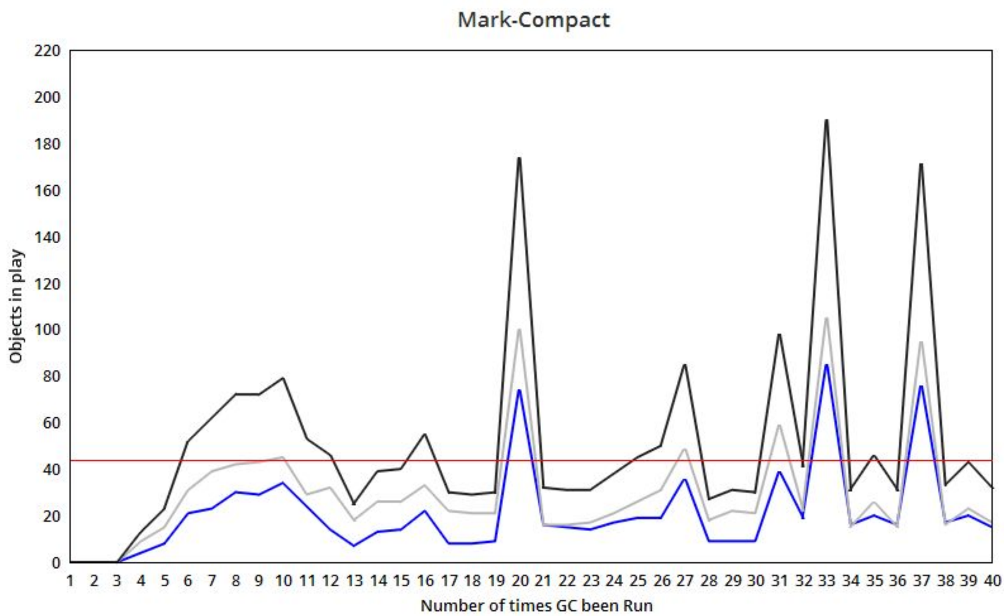




The average FPS for the reference counting is 36 FPS and at 39 as an average without the first five iterations which is well below the required 45. It is not able to run on an FPS above 45 FPS but it actually reaches 44 several times. The average number of objects in play is 25.

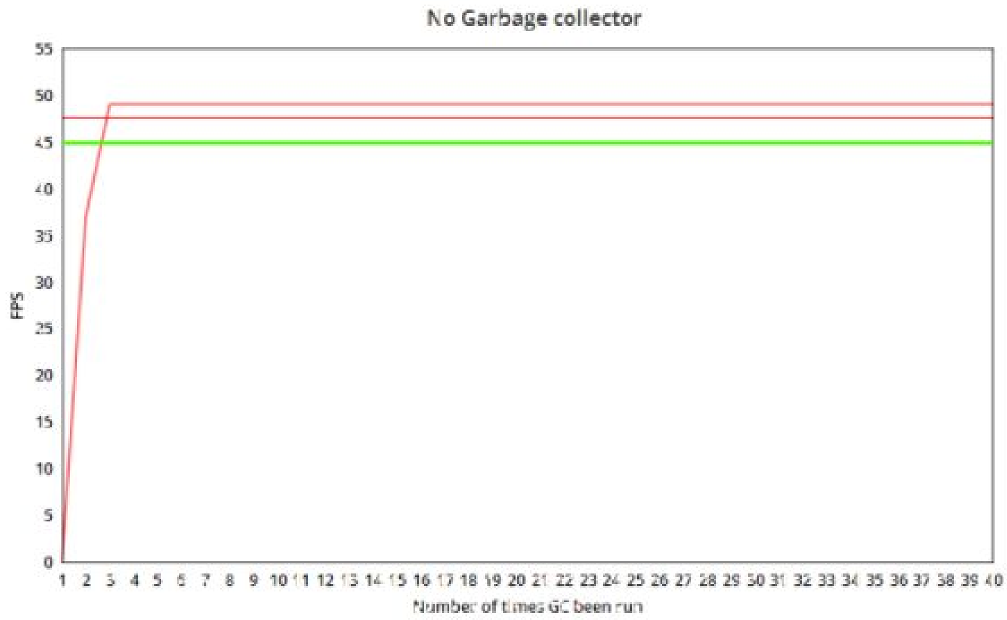
5.2.3 Mark-Compact Garbage Collector

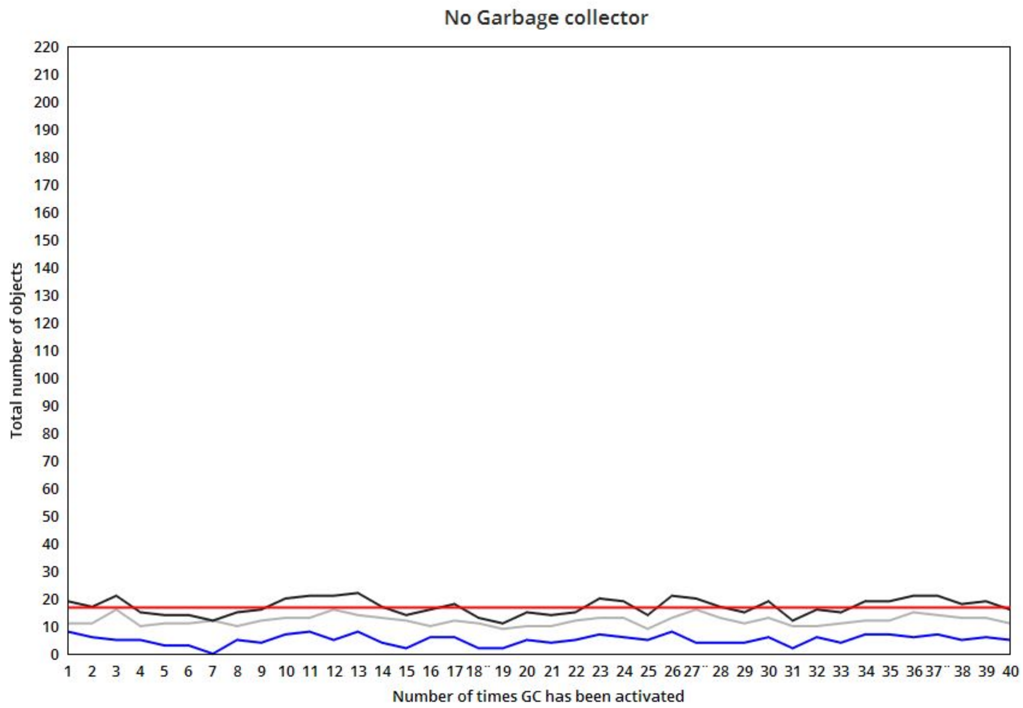




The mark compact algorithm is running with an average of 36 FPS and at 38 as an average without the first five iterations, putting its average below the recommended 45, it also has the issue that it is not able to reach 45 FPS at all. The average number of objects in play is 50 objects when cleaned up by the Mark compact garbage collector. As the graph above shows there are some times when they peak before the garbage collector is able to bring the items down again.

5.2.4 No Garbage Collector





The average fps is 47 and at 49 as an average without the first five iterations. With no garbage collector there are an average of 17 objects in game at any given point.

6 Analysis

Regarding fulfillment of requirements, the use of a garbage collector fulfills requirement 1.1 ("do not run out of memory") and the design of the game (as described in chapter 4) fulfills Requirement 1.2 ("stall the program less than 100 ms each time the GC is run"). This leaves requirement 2 ("keep framerate above 45 fps"). None of the garbage collectors are able to keep the FPS either average or individually at the required 45 FPS. However, the reference counting and Mark compact both run at 36 while mark sweep ran at 35. Despite this the version without a garbage collector can run at only 47 and also due to the simplistic nature of the game, it is able to keep a very consistent level of framerate. As far as objects in play are concerned, the Mark Sweep does have the highest average number of objects in game while the Mark Compact holds a similar level of frames. The reference counter however is able to keep the number of only half of that at an average of 25 objects.

No garbage collector is able to keep the number of objects compared to the manual version that only has an average of 18 objects in play at the same time. We can see ups and downs in the graphs but the extreme peaks generally tend to be in similar positions on the X axis. This most likely points to where the ship changes direction, since it is already traversing positions where it has destroyed the asteroids. This means that it will not spawn new bullets and allowing the garbage collectors to work through the backlog. Similarly larger build ups would occur when the ship reaches an asteroid that just spawned since there is a larger distance for the bullets to travel. This means that the garbage collector will not mark the salvo until later and there are more objects to update. There are also differences between the various garbage collectors which is to be expected since they approach garbage collection in different ways.

Despite how big the differences in objects are, they are not enough to affect how many frames per second the game is running at. Both the reference counting and mark compact are running at 36 FPS while Mark sweep is running at 35 FPS. All three have similar number of objects to compute to not result in any noticable drops in framerate. The effectiveness of the garbage collectors themselves is what results in the differences of objects. For example while the mark sweep garbage collector needs to loop through the objects twice to be able to delete them the reference counting garbage collector is only forced to do so once. This results in it having a much lower number of objects in play than either the mark compact and the mark sweep. Similarly the more efficient way that mark compact saves information results in a lower number of objects than the more primitive mark sweep garbage collector. However the game without the GC was clearly superior running both at the required framerate and with less objects in play. Having fewer objects and with less extreme differences between each time the frame rate is calculated results in a more stable framerate as well.

We can also see how the mark compact compares to the mark sweep in keeping the game running with less objects and with more stable ups and downs as far as number of objects are concerned. While the objects in this simulation were very similar in design and the amount of memory they took up the mark compact's operation only resulted in an average of four objects less and one frame per second higher framerate compared to the mark sweep. However, this gap continue to increase if the game would use more complex objects where the amount of data stored differed widely between each asteroid or bullet. The fact that the version without a garbage collector is more effective in this version is not too surprising because of the simplicity of the game. If one were to make a larger game it would be necessary to take the time it would cost to make sure there are no drains into account and this is highly dependant on the developer and project.

7 Conclusions

If a developer is able to manually deal with memory gaps this is superior both frame and object wise to do, rather than relying on a garbage collector. But this also means the developer would need the time and resources to make sure the program does not have any memory gaps. As such a garbage collector would most likely be better for a game that used a lot of procedurally generated content. Another good solution for where a garbage collector might find use is a game that has a lot of content that cannot be checked manually, like an MMORPG game where the developer might just lack the time and resources to make sure that the game lacks any memory gaps.

One cannot always expect to have such optimal conditions to work in and out of the three versions tested in this thesis the reference counting garbage collector is the most effective both in regards to the lowest amount of objects and framerate. This is however in a test that is a rather extreme and does not take full advantage of a Mark Compact Garbage Collector. A fairer test would be a test that used asteroids that each took a different amount of memory space during a longer period of time. This would give the mark compact garbage collector a better chance to prove its strengths according to the parameters of this thesis. An example of this is to include a future test where the asteroids has derived classes that each has different properties as well as different kinds of bullet classes that is spawned randomly. What these differences might be is not relevant for the discussion. However them simply existing will give the computer different sized files of data that leaves different sized gaps to be filled in, in the memory. This may result in a test where the mark compact will have a more noticeable advantage. However that is for a future study and is not covered in this paper. What the tests shows, however, is that the simplistic approach in the mark sweep is not worth it compared to the more advanced garbage collector as it lags behind not only in the total amount of frames per second but also in the amount of objects in play.

As the test is designed, the garbage collectors are hindered by the extreme pressure put on them due to the large amount of objects spawned and the poor computer they had to work with. All of these problems could be countered with a more efficient computer and while it is hard to say exactly how much better a computer would need to be to actually run at the desired framerates should any modern computer be capable of running at the desired framerates. For instance running the test at an asus ROG GL552JX laptop are the garbage collectors were capable of handling the tests at the desired framerate.

In conclusion; the garbage collector would be better to use for larger games. However for a smaller development studio working on a smaller program in the indie industry like what Daniel Benmergui described in his tweet a garbage collector most likely caused more trouble than it solved.

References

http://www.citethisforme.com/harvard-referencing

Apple Inc. (2011). *Developer Tools Kickoff - session 300*. [video]. Available at:
https://developer.apple.com/videos/play/wwdc2011/300/
[Accessed: 27 Sep. 2016]

Benmergui, Daniel. (2015). *In order to optimize my game, I have to go out of my way to prevent the Garbage Collector from doing anything. I wish I could turn it off.*[Twitter]. 18 April. Available from: https://twitter.com/danielben [Accessed: 8 Sep. 2015]
Link to actual tweet: https://twitter.com/danielben/status/589259667827744768
Link to the game: http://ernestogame.com/

Card, S., Robertson, G., and Mackinlay, J. (1991). The information visualizer: An information workspace. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. [online] New York: ACM, 181-188. Available at:
http://www2.parc.com/istl/groups/uir/publications/items/UIR-1991-01-Card-CHI91-IV.pdf
[Accessed: 15 Aug. 2016]

Claypool, M. and Claypool, K. (2009). Perspectives, Frame Rates and Resolutions: It's all in the Game In: *Proceedings of the 4th International Conference on Foundations of Digital Games*. [online] New York: ACM, 42-49. Available at:
http://web.cs.wpi.edu/~claypool/papers/perspective/paper.pdf
[Accessed: 10 Sep. 2016]

Fong, A. and Richard, C. (2002). Dynamic Memory Allocation/Deallocation Behavior in Java Programs. In: *2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering*. [online] IEEE, 314-317. Available at:
http://ieeexplore.ieee.org.proxy.mah.se/stamp/stamp.jsp?tp=&arnumber=1181277
[Accessed: 8 Jul. 2016]

Fox, Ken. (2014). *Visualizing Garbage Collection Algorithms*. Available at:
https://spin.atomicobject.com/2014/09/03/visualizing-garbage-collection-algorithms/
[Accessed: 3 Sep. 2016]

Hertz, M. and Berger, E. (2005). Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In: *OOPSLA '05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. [online] New York: ACM, 313-326. Available at: http://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf
[Accessed: 8 Aug. 2016]

Janzen, B. and Teather, R. (2014). Is 60 FPS Better than 30? The Impact of Frame Rate and Latency on Moving Target Selection. In: *CHI '14 Extended Abstracts on Human Factors in Computing Systems*. [online] New York: ACM, 1477-1482. Available at: http://www.csit.carleton.ca/~rteather/pdfs/Frame_Rate_Latency.pdf
[Accessed: 10 Sep. 2016]

Miller, R. B. (1968). Response time in man-computer conversational transactions. In: *AFIPS '68 (Fall, part I) Proceedings of the December 9-11, 1968, fall joint computer conference, part I.* [online] New York: ACM, 267-277. Available at: http://theixdlibrary.com/pdf/Miller1968.pdf
[Accessed: 15 Aug. 2016]

Myers, B. A. (1985). The importance of percent-done progress indicators for computer-human interfaces. In: *CHI '85 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. [online] New York: ACM, 11-17. Available at: http://delivery.acm.org/10.1145/320000/317459/p11-myers.pdf?ip=31.208.201.27&id=317459&acc=ACTIVE%20SERVICE&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E0B34975157C245C7%2E4D4702B0C3E38B35&CFID=826638587&CFTOKEN=49027773&__acm__=1471421929_e4e95344e7b0a6056aafa76817aac5c7
[Accessed: 15 Aug. 2016]

Padron-McCarthy, Thomas (2008). *The Very Basics of Garbage Collection*. Available at: http://basen.oru.se/kurser/koi/2008-2009-p1/texter/gc/index.html
[Accessed: 20 Jul. 2016]

Priess, Bruno. (1998). *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Available at: http://www.brpreiss.com/books/opus5/html/page428.html
[Accessed: 3 Sep. 2016]

Ravenbrook Limited. (2016) Memory Management Glossary: garbage collection
Available at: http://www.memorymanagement.org/glossary/g.html#garbage.collection
[Accessed: 5 Nov. 2016]

Ritzau, Tobias. (2003). *Memory Efficient Hard Real-Time Garbage Collection*. Available at: http://liu.diva-portal.org/smash/get/diva2:20899/FULLTEXT01.pdf
[Accessed: 19 Aug. 2016]

Schoeberl, M. and Puffitsch, W. (2010). Nonblocking Real-Time Garbage Collection. In: *ACM Transactions on Embedded Computing Systems (TECS)*, [online] Volume 10(1), Article 6. Available at: http://www.jopdesign.com/doc/nbgc.pdf
[Accessed: 12 Aug. 2016]

Thomas. Paul (2010). Incremental Parallel Garbage Collection
Availible at: http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/p.thomas.pdf

[Accessed: 24 Nov. 2016]

Thompson, Martin. (2013). Java Garbage Collection Distilled
Available at: https://www.infoq.com/articles/Java_Garbage_Collection_Distilled
[Accessed: 24 Nov. 2016]

Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian (2008)
Selecting Empirical Methods for Software Engineering Research
[Accessed: 10 April. 2017]
http://ieeexplore.ieee.org.proxy.mah.se/stamp/stamp.jsp?tp=&arnumber=1181277

Appendix

Testing Environment

The tests are performed on a 5 year old HP probook 4535s where the program is also programmed, executed, and measured with Windows Enterprise Visual Studios 2015.
The specifications of the laptop computer:

- Processor: AMD A6-3420M APU with Radeon(™) HD Graphics 1.50GHz
- Installed RAM: 8 GB
- OS: Windows 10 Pro, 64-bits

Tests are also performed on a considerably more powerful computer.
The specifications of the desktop computer:

- Processor: Intel(R) Core(™) i5-3570K CPU @ 3.40GHz
- Installed RAM: 12.0GB
- OS: Windows 10 Pro, 64-bits