WORKSHEET D

Version 1.0
Computing
COMP270

Kate Bergel

Introduction

In this worksheet you will make use of the profiling tools in Visual Studio[1] to analyse the perfomance of a simple game, using the results to guide your efforts to improve efficiency.

Begin by **forking** the following git repository:

```
https://github.com/Falmouth-Games-Academy/comp270-worksheet-D
```

Complete the tasks described below, remembering to **commit** your work regularly. To submit your work, open a **pull request** from your forked repository to the original repository.

Notes:

- Completing the worksheet requires writing answers to the questions posed, as well as implementing code changes. The written answers should be fairly short, so can be included in the comments, git commit messages, or readme file.
- Profiling results should also be included to show the effect of your code changes. If you can export the data as a text file, it may be included in this format; otherwise, write down the key information in your answers/-comments.
- Some of the implementations in the tasks below may be concentrated on a particular function or data structure, however you are free to modify any parts of the code you think appropriate. Make sure to justify your changes in code comments and/or git commit messages.
- Tasks 2 and 3 can be implemented in any order, and task 4 may be attempted without their being completed, but task 1 *must* be completed first.

Background

The code provided implements the basic features of an asteroids game: manoeuvring a "spaceship" around the screen and firing bullets at space rocks to break them into smaller fragments, as shown in Figure 1. Features such as scoring and health/player lives are not implemented as these are not necessary for (and would possibly interfere with) the main focus of the tasks below - which is to make sure that the game will continue to play smoothly and with the best frame-rate possible, even with large numbers of objects.

The game uses the following keyboard controls:

- `up arrow`: applies a thrust in the direction the spaceship is facing
- `right/left arrow`: rotates the spaceship clockwise/anticlockwise
- `space bar`: fires a bullet from the front of the spaceship

[1] Or similar tool of your choice.

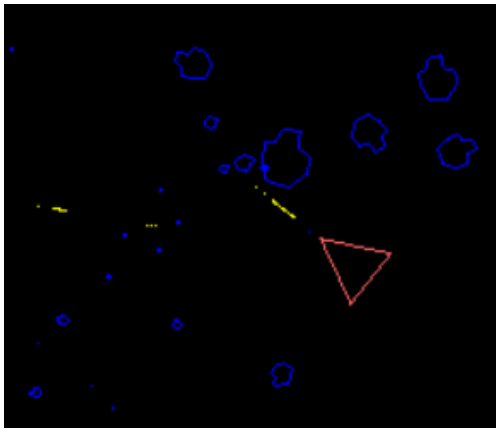

Figure 1: The game in action

Task 1: making your benchmark

(a) The main bulk of the code runs in `Application::update()`, which handles the game logic, and `Application::render()`, which deals with the drawing. **Read** through these functions, and the ones they call (directly or indirectly), looking for obvious inefficiencies.
List any that you spot, and **state** how they might cause problems. Where do you think most of the work will be done?

(b) Now build the code and play through the game for a few minutes, shooting as many asteroids as you can. **Write down** any observations about the performance or other issues, and potential causes; these form hypotheses to test using profiling.

(c) In order for benchmarks to be useful, they should be run on a reproducible test case. Since user input is likely to vary between runs, **modify** some of the values[2] and add logic to create a scenario that will push the code by generating a reasonably large number of objects, with minimal input from the player[3]. Make sure that your test case will be representative, and call all the game code[4].
Commit your changes, so that you can use this test case in the remaining tasks.[5]

(d) Run the Visual Studio CPU profiler (or other tool of your choice) to measure the times taken by the various functions. **Examine** the results and **state** which functions[6] are using the largest percentage of time. Are they the ones you predicted? If not, **explain** what might be causing the results.

(e) Now run the memory profiler, remembering to take snapshots at the start and regular intervals. Do these results match your expectations? **Explain** why (or why not).

Task 2: insider knowledge

In order to make the asteroids explode at an appropriate time, the code has to check whether any bullets are inside each asteroid's boundary. This kind of collision detection is often one of the most computationally intensive parts of a game.

---

[2] Hint: the asteroid spawning settings are in `Application.h`.

[3] Ideally none.

[4] Including, for example, breaking the asteroids into pieces.

[5] Normally, you should make sure to remove any such 'hacks' before committing your final solution, but don't worry about that for this assignment.

[6] Of the ones in the actual game code only - ignore SDL and any other 3rd party or Windows native code.

The algorithm implemented in `Asteroid::pointIsInside()` is known as the "angle summation test", but there are several alternative approaches to testing whether a point is inside a polygon, as described here: `https://erich.realtimerendering.com/ptinpoly/`

(a) See if you can increase the efficiency of `Asteroid::pointIsInside()` by **implementing** an alternative algorithm. **State** which one you have chosen and justify your choice in the comments.
Re-run the CPU profiler on your test case and **state** how much the time has improved.

(b) Can you improve the average efficiency of this function even further by reducing the number of times the full point-in-polygon test is carried out? **Implement** an "early exit" condition to avoid testing bullets that are "obviously" far away from the asteroid, then **re-run** the CPU profiler again to see if it has helped.

Task 3: space debris

Both the asteroids and bullets are stored in `std::vector` containers, with new objects being added using `push_back()` whenever the space bar is pressed, an asteroid is shattered or a new one is spawned in `Application::update()`.

(a) Thinking about the CPU and memory profiling results from Task 1, **list** the pros and cons of the current implementation using `std::vector`.

(b) Use the debugger, or a manual counting mechanism, to keep track of how many objects are in the scene. **Compare** this number to what you can see on the screen - do the figures correspond? **Explain** why this is and how it might be affecting the performance.

(c) **Describe** two or more different ways you could compensate for and/or reasonably limit the large number of objects that are being generated, without affecting the game experience[7].
Might using a different data structure be more appropriate? Which one, and/or why (not)?

(d) **Implement** (at least) one of your ideas above, testing the effects on perfomance as usual.

Task 4: how low can you go?

(a) **Examine** your latest benchmarks (re-running them if necessary) to see which other operations are taking up a significant proportion of the time[8]. **List** which of these you think it might be possible to improve upon.

(b) **Implement** at least two changes to address one or more inefficiencies, **explaining** your motivations in the comments/commit messages and noting the results of profiling after each one.

(c) **Describe** any further enhancements you think might be beneficial to the game's performance.

[7] Ideally keeping the asteroids spawning at a consistent rate for the duration.
[8] You may get more information by profiling in debug mode.

Marking Rubric

Criterion	Weight	Refer for Resubmission	Adequate	Competent	Very Good	Excellent	Outstanding
Basic competency threshold	30%	A reasonable attempt at the worksheet was not submitted by the formative deadline.	A reasonable attempt at the worksheet was submitted by the formative deadline. There is no evidence of academic misconduct.				
Functional coherence	40%	None of the tasks have been attempted.	Task 1 has been attempted and partially completed.	Task 1 has been successfully completed.	Two tasks (including task 1) have been successfully completed.	Three tasks (including task 1) have been successfully completed.	All four tasks have been successfully completed.
Maintainability	30%	The code is only sporadically commented, if at all, or comments are unclear. Few identifier names are clear or inappropriate. Code formatting hinders readability.	The code is well commented. Some identifier names are descriptive and appropriate. An attempt has been made to adhere to a consistent formatting style. There is little obvious duplication of code or of literal values.	The code is reasonably well commented. Most identifier names are descriptive and appropriate. Most code adheres to a sensible formatting style. There is almost no obvious duplication of code or of literal values.	The code is reasonably well commented, with appropriate high-level documentation. Almost all identifier names are descriptive and appropriate. Almost all code adheres to a sensible formatting style. There is no obvious duplication of code or of literal values. Some literal values can be easily "tinkered".	The code is very well commented, with comprehensive appropriate high-level documentation. All identifier names are descriptive and appropriate. All code adheres to a sensible formatting style. There is no obvious duplication of code or of literal values. Most literal values are, where appropriate, easily "tinkered".	The code is commented extremely well, with comprehensive appropriate high-level documentation. All identifier names are descriptive and appropriate. All code adheres to a sensible formatting style. There is no duplication of code or of literal values. Nearly all literal values are, where appropriate, easily "tinkered".