Research and Quality Assessment Exam
Speed evaluation between C# and C++

Kristofer Pedersen
cph-kp278

Mads Knudsen
cph-mk682

Michael Frederiksen
cph-mf315

Søren Merved
cph-sm428

February 17, 2025

Abstract

This report is an investigation in the comparison of computational efficiency of C# and C++ algorithms in optimizing navigation meshes for video game environments. This project stems from the general trend of higher demand for video game performance. As the video game engines Unity and Unreal (which utilize C# and C++ respectively), gives this project significant relevance for developers in the gaming industry. By analyzing various datasets, compromised of navmeshes generated in Unity, we aim to empirically assess our hypothesis, that C++'s closer to hardware compilation and manual memory management would argue for a better performing language. Despite initial expectations of C++ being universally faster, our findings suggest a more nuanced result. C# demonstrates a faster processing speed, especially for smaller navmeshes and our results suggest that C++ will perform quicker on larger navmeshes.

Contents

1 Introduction

Speed comparisons are not uncommon among code languages but they are primarily made around small lines of code, where we seek to do a comparison using a larger algorithm. The algorithm we have chosen has been selected because of its scale and complexity, as well as that it was made with the intent to be part of a released product.

In this report, we will be investigating the computational speed of C# and C++ in regards to computing Navigational mesh optimization. Navigational Meshes (Navmesh) are highly relevant in video gaming and other applications that involve autonomous character movements. A navmesh represents the walkable ground, which an autonomous bot or AI agent is able to walk on, in a virtual environment.

The choice of programming languages is very relevant in the video game industry. Unity is a versatile and powerful engine and widely used for its easy accessibility by developers and uses C# for scripting. Unreal engine is regarded as one of the most powerful and advanced gaming engines and it is known for its cutting edge graphics with real time rendering. Unreal is developed using C++. These two game engines alone dominate 33% of the steam market share, the biggest online market place for PC video games. This suggests that there is a large demographic of C# and C++ developers in the gaming industry which is the basis for why these languages were chosen. [1]

2 Hypothesis

Given the critical role of navmesh optimization in how non-player characters are able to move in virtual environments, the choice of programming languages is important and will have an overall affect on the computation time. We hypothesise that C++ is faster at computing than C#. C++ is known for its high performance and close-to-hardware programming capabilities and compiles code close to machine code. The direct compilation can lead to highly executable programs that are specific to hardware. C# compiles to a CLR which is an intermediate language. This introduces an extra layer between the program and the hardware which results in slower processes. C++ is known for its manual memory allocation and deterministic detractors. Memory leak can cause slower performance, but if implemented correctly, it allows for optimization that can significantly reduce memory usage and speed. In contrast C# relies on its own garbage collection management. A simpler solution for development, but slower performance wise. [2]

When considering the outcome of this report, we are purely basing the performance on compute time. Based on this metric we expect C++ to outperform C#. [3]

3 Code

3.1 Algorithm Origin and Description

Before we started working on this project, one of our group members had been working on a side project, where the build-in solution for an AI agent navigation in Unity did not meet the requirements for the end goal. The navmesh produced, would only include walkable areas, area costs and offmeshlinks, which were used to enable the agents to jump. The navmesh also only supported a relatively small number of different agent sizes, which would not work with 100+ different types of agents, and lastly it would also produce vertices which could be considered overlapping each other. [4] [5]

The optimization algorithm was created to first reduce the number of vertex points, by checking each points distance to the other points. Any point found to be overlapping would then be removed and any triangles it would have been a part of, through the indices list, would have it replaced by the original, which it was checked against.

It would then create a list of navigation triangles based on the indices list, where each triangle included the IDs of its neighbors. It would then use a flood fill algorithm[6], triangle to triangle based

on their neighbors, where the first triangle in the search would be selected by being the closest to a manually placed 3D point, called a clean point. A new list of vertices and indices would then be created from the resulting triangle list.

To ensure that there aren't any holes in the navmesh, it would then check each vertex and whether two of its neighbours were also connected to each other, and that there was not already an existing triangle in the indices list made up of those three. It would then add those three to the indices list. Lastly it would go over the new vertices and indices and create the final list of navigation triangles to store.

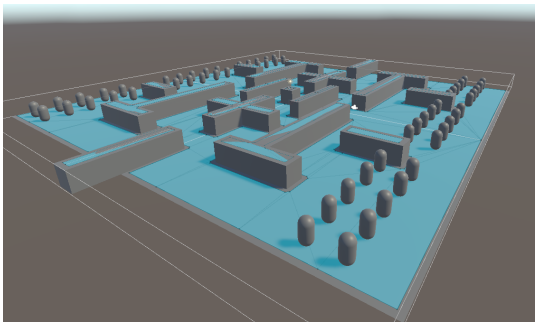

Figure 1:
Navmesh generated by Unity with unreachable areas on top of the boxes.

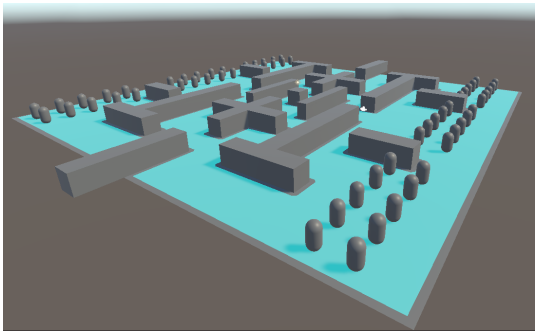

Figure 2:
After optimization without unreachable areas. Reduced vertex and indices count.

3.2 Refactoring

Since the original algorithm was written in C# for Unity, most of the code didn't need much change when writing it as a standalone version, but there were some Unity specific libraries which needed replacements. These changes were primarily around Unity's mathematics library and their Vector2, Vector3 and Vector2Int structs. All of which were easy to refactor as the source code could either view or recreate by looking up the math online.

With C++ we needed to rewrite the whole thing from scratch. With most things used within the C# code, there were a equivalent function and libraries already available in C++, lists for C# and vector for C++ etc. The biggest difference between the two are the handling of memory. Unlike C#, the C++ code needed to also manage the creation, reading and changing of variables.

Both make use of external libraries for reading json files of the unoptimized navmeshes generated and exported from Unity with an editor script we wrote.

3.3 Usage

Both versions of the algorithm was run in release mode through their respective IDE on the same computer. Both scripts would run three for-loops, one nested within the other. The first loop would select the category, Small, Medium and Large (S, M and L), based on the area size used to generate the json file. The second loop would select a number, 1-5, which represents a particular file in that category. The purpose of the last loop was to run each test set a thousand times for a larger data sample result. A timer would be started and then the very next line would run the algorithm, after which the time in milliseconds would be recorded. Lastly the results would be stored in a csv file for easier use for data analysis.

4 Data evaluation

To illustrate the impact of the optimization algorithm, we have plotted the number of vertices and indices, before and after it has been run through the optimizer algorithms. Figure 3 shows the vertex count. The C++ and C# algorithms have similar results in how they reduce the number of vertexes. The optimized of vertex count has been reduced to an average of 19.5% of the original number. The indices figure (figure 4) paint a similar picture. C# and C++ result in the exact same number of indices, and have reduced the number by 50%.

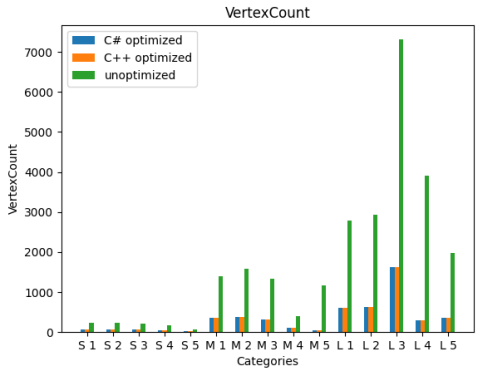

Figure 3: Vertex count before and after optimization

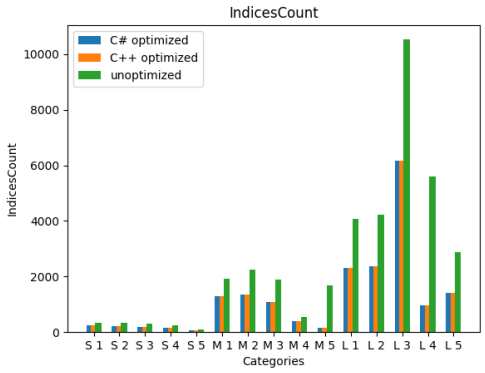

Figure 4: Indices count before and after optimization.

To evaluate the algorithms, we ran the optimizer on each navmesh a total of 1000 times. For each run, we recorded the computational time (in ms), as well as the vertex and indices count after optimization.

Figure 5 shows the average time for all of the runs. The plot shows that there is no clear optimal language. For the most part they both have similar performances, and on most datasets the differences are minimal. Even the difference is so small, C# outperforms C++ on most datasets.

C++ is only able to outperform on 3 of the datasets: L1 , L2 and L3. This is however interesting as L 1, L 2, and L 3 are the largest datasets in regards to vertex and indices (see figure 3 4). This could suggest that C++ performs better on larger datasets.

When analysing the data, we can see that the small size navmeshes are so small, that their computational time is approximately 1ms. This is an issue because the minimum unit for our timer is 1ms. This means we won't be able to get accurate results when considering these smaller navmeshes.

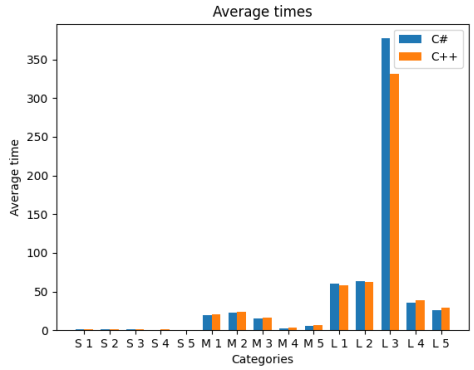

Figure 5: Average time for each navmesh dataset.

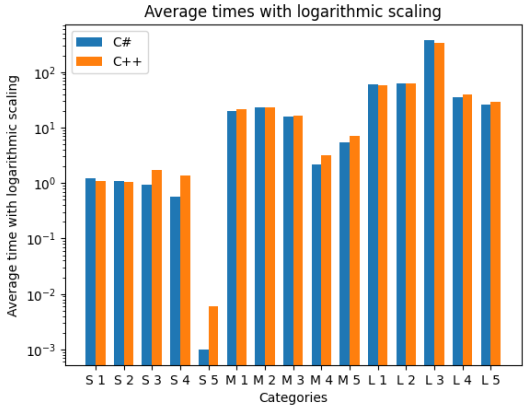

Figure 6: Average logarithmic time for each navmesh dataset.

To understand the behaviour we are seeing from the results of the algorithm we will look at the data it needs to go over. With the datasets we can see from figures 7 and 8 that the time to complete for the algorithm increases exponentially as the vertex count and indices count increase. This is also what we expected, as the algorithm includes multiple nested for loops where one vertex would check the position of other vertices.

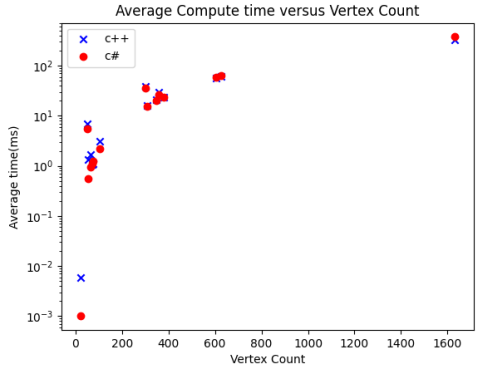

Figure 7: Compute time in logarithm versus vertex count

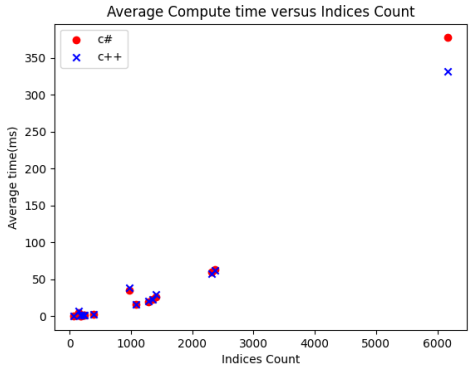

Figure 8: Compute time versus indices count

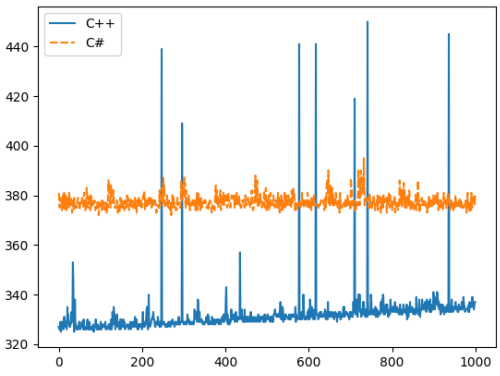

Figure 9: Plot for the times for 1000 test runs for set L 3

Out of all the test sets we have generated, the size of L 3, see figure 3 and 4, allows us to view a bigger difference of time to complete produced between C# and C++.
While C# proved to be faster on the smaller datasets, C++ proves to be faster in the largest dataset. While C# is fairly consistent across the test, C++ shows massive spikes where it shows to be slower than C#, see figure 9.
Most importantly we can see that C++ slowly tend to slow down the longer it runs. Looking at the line for C++, we can see a general upward trend in times. As we run the algorithm more and more times, the C++ slows down. While we don't have any data that might specify the reason as to why this happens, we can make a guess, since the C++ code isn't fully optimized and likely to contain some memory leaks.
This would explain why the program runs slower the longer it runs and why we can best see this effect with the largest dataset, as more data means a bigger likelihood of a leak.

5 Conclusion

From our chosen algorithm we have run multiple datasets containing different navigational meshes, and from those test runs we have gathered and evaluated data. From that data we could not prove our set hypotheses that C++ is faster than C#, with the data showing C# to be faster for smaller datasets, while C++ shows to be faster for the largest datasets, and otherwise they show to be similar.

We can, however, not give a definitive answer, as we found during our evaluation of the data that the C++ code appears to not be properly coded and contain improperly managed memory.

References

[1] M. Toftedahl, "Which are the most commonly used game engines?," *GameDeveloper*, 2019.

[2] Microsoft, "Garbage collection," *.Net Documentation*, 2021.

[3] Y. Shiotsu, "C# vs. c++: Which language is best for your project?," *Upwork*, 2024.

[4] Unity, "Building a navmesh," *Unity Documentation*, 2023.

[5] Unity, "Creating an off-mesh link," *Unity Documentation*, 2023.

[6] FreeCodeCamp, "Flood fill algorithm explained," *FreeCodeCamp#Algorithms*, 2020.