The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA
School of Computer Science

4288618

Supervisor: Dr Jason Atkin

Module Code: G53IDS

2019/04

Comparison of some occlusion culling algorithms used in modern game engine development

Submitted April 2019, in partial fulfilment of
the conditions for the award of the degree **Bsc (Hons) Computer Science.**

Harry Stephen James Hollands

4288618

School of Computer Science

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature _____ _____

Date __11__/__04__/_2019_

I hereby declare that I have all necessary rights and consents to publicly distribute this dissertation via the University of Nottingham's e-dissertation archive.*

Abstract

This project involves the use of research of the Unity engine and algorithms often used in game engine development. Two additional game engines have been written based upon an existing engine that I have written from scratch. One engine shall contain algorithms known to be used in Unity, and the other shall contain external algorithms which may yield improvements over the existing Unity techniques. Both engines shall be analysed in multiple ways to deduce whether these newer external algorithms may yield an improvement when used in scenes and scenarios expected in a real game application.

Table of Contents

Introduction

Game engines are among the first major choices made during the early stages of game development. From id Tech 1 (Doom and Doom 2) to Unreal Engine 4 (Fortnite, Gears of War 4 and many more...), some of the most popular and iconic video games were all driven by exceptional game engines. Engine technology has progressed massively throughout the decades; particularly the many algorithms which drive their functionality.

The aim of this project is to compare and contrast one or more of the most common techniques and algorithms used in the popular Unity game engine with one or more external algorithms to deduce whether such algorithms could be

an improvement over those used by Unity and thus may be better technologies for game development. The aim is not to deduce improvements mathematically beyond a doubt; this is a process too subjective to achieve. One of the methodologies of this research project is identifying exactly how an algorithm can be an "improvement" over another. Once attributes are identified, analysis is performed on both algorithms to gather metrics relating to these attributes in an effort to conclude under which circumstances the algorithm could be preferred.

This project shall deduce whether they are superior in the general case of game application runtime. In many cases, superiority in the general case cannot be justified, as there may not be enough evidence gathered to account for all cases. In this scenario, identifying under which circumstances the algorithm may be superior becomes the primary goal. The objectives that shall be met to achieve this aim are as follows:

- To write a new game engine based upon the existing Topaz engine (named Engine A) utilising one or more algorithms chosen from Unity.
- To improve Engine A with newer algorithms and technologies which could improve the performance, accuracy and/or flexibility of the engine in a game application (Engine B).
- To contrast the specific performance, accuracy and flexibility of each new technology in Engine B with the previous technology used in Engine A.
- To analyse the contrasting performances, accuracies and flexibilities of the algorithms to deduce whether the algorithm used in Engine B could be a better alternative than the algorithm used in Engine A/Unity under conditions expected in a game application.

It is necessary to highlight that the algorithm utilised by Unity is compared only, meaning that additional functionality provided by their implementation of the algorithm shall not be reimplemented.

Motivation

Virtually all modern games use an existing game engine. For this reason, many different engines exist under both proprietary and open-source frameworks.[1] Game studios often use their own private game engines, but the majority of game developers use a free license from famous public game engines, such as Unity and Unreal Engine.[2] This means that a handful of game engines drive the majority of games on the market. For this reason, it is essential that these game engines utilise algorithms with high performance, accuracy and flexibility. When new algorithms are innovated which could be superior to existing algorithms in game engines, the developers of these engines would be prudent to consider updating their engine to have these improvements. Unity has done this before with a Pure ECS addition to the engine.[3]

However, new algorithms are often discovered and these engines often fail to improve their systems accordingly. Because of this, algorithms which could greatly improve the performance of games are not taken advantage of, preventing games from being improved as rapidly as they could. There are a multitude of reasons as to why engines such as Unity do not assimilate these algorithms. One of these reasons is because the success of these algorithms are not proven in realistic game scenes.

Related work

When new algorithms are innovated, there is often much research completed in order to provide evidence of attributes of the algorithm, such as performance and flexibility. Often, the innovators of these algorithms provide evidence of

[1] Wikipedia maintains a useful list of game engines: *https://en.wikipedia.org/wiki/List_of_game_engines*
[2] Unity claims via their public-relations page that over 3 billion devices use it worldwide, and is used in over half of all mobile games. *https://unity3d.com/public-relations*
[3] Unity introduces a new C# Job System along with a new ECS *https://unity3d.com/unity/features/job-system-ECS*

such properties. However, the evidence is often based in isolated cases. Excerpt from the Unreal Engine Overview on performance and profiling:

> "Performance is an omnipresent topic in making real-time games. In order to create the illusion of moving images, we need a frame rate of at least 15 frames per second. Depending on the platform and game, 30, 60, or even more frames per second may be the target. Unreal Engine provides many features and they have different performance characteristics. In order to optimize the content or code to achieve the required performance, you need to see where the performance is spent. For that, you can use the engine profiling tools." (Epic Games. Performance and Profiling Overview)

Popular game engines such as Unity always offer their own profiling technology to the user[4]. However, there are other important properties to game engine technologies than performance alone. This project focuses not only on performance, but also on accuracy and flexibility so to better judge the capabilities of the chosen algorithms.

Methodologies

Preliminary Research

The aim of the project is to compare and contrast algorithms in a manner relevant to game development. To do this, it is necessary to decide which properties of the algorithms should be compared. In other words, which properties of algorithm X and algorithm Y could be contrasted to deduce whether one is 'superior' to another? The research carried out yielded the following three properties to be the most essential for justifying the use of a particular algorithm in a game engine:

Performance

Performance is the most obvious property which is important for a game engine. It is a property of higher value in game development than it is in other technical fields. This is because games are often substantially more demanding than other applications because of engine technologies providing features such as 3D graphics, constant physics simulation and other demanding tasks.

If algorithms in a game engine perform poorly, then the games running on the engine may have unacceptable performance issues such as low or volatile frame-rates and latency. This is unacceptable and will not be tolerated by the users of such games.

Accuracy

Algorithms exist to solve a given problem. Accuracy of an algorithm is the extent to which it truly solves the problem that it was intended to solve. Accuracy is a property important universally across many technical fields; a training algorithm that fails to improve the fitness of a neural network is disastrous in a machine learning application, similarly to how a collision detection algorithm which fails to accurately detect collisions between objects is disastrous in a game application. Although all of the three highlighted properties of algorithms are universally required to an extent, accuracy is a property that benefits all use-cases; an inaccurate algorithm, no matter how flexible or performant, is unable to fulfill its purpose because it did not provide the correct output/functionality.

[4] Unity documentation on the profiling tool. https://docs.unity3d.com/Manual/Profiler.html

Flexibility

> "The first, most important feature of a middleware package is its integration complexity. Good libraries are highly modular, minimally intrusive, and easy to plug into very different codebases. In short, they make very few assumptions and are fairly decoupled from implementation details of other systems." (Lengyel E., 2010)

This excerpt from 'Game Engine Gems, Volume One' gives rise to the significance of the property of flexibility. A game engine utilises a multitude of algorithms. Utilising inflexible algorithms, however, hinders the overarching quality of the engine as it imposes restrictions and limits on what functionality the engine is able to offer under certain conditions, which is a major disadvantage for a product designed to accommodate a multitude of genres.

Unity Research

It is imperative to carry out research to understand the underlying technologies driving Unity. This is because the algorithms that exist in Engine A must also exist in Unity and be among those chosen by this research. Such technologies shall consist of algorithms utilised by Unity for its various processes, such as graphics and physics processing. Unity has a manual available which discloses a multitude of the algorithms it uses and explanations of how they work. Using this manual is of paramount importance because Unity is not an open-source project; there is no other reliable way of identifying and understanding the algorithms used in the engine. It is through the Unity manual that the entirety of the Unity research is carried out.[5]

External Algorithmic Research

It is not sufficient to merely establish an understanding of only the technologies driving Unity if the objectives outlined earlier are to be met. It is necessary to identify and implement algorithms that are not used by the Unity engine. These external algorithms must correspond to a Unity algorithm identified during the Unity research. In other words, if Algorithm A in Unity solves Problem X, then an external Algorithm B must also solve Problem X. Specific strategies must be utilised to locate and identify external algorithms. These strategies include:

- Identifying algorithms used in other popular game engines such as Unreal Engine 4 which differ to the ones used in Unity.
- Viewing existing technological documents such as books and academic papers which propose alternative solutions to problems which are solved in game engines.

Game Engine Development Processes

Two game engines are developed throughout the project; Engine A and Engine B. These engines shall eventually be analysed and closely compared with one another. The development processes of the two engines should be similar, as both require implementations of different algorithms obtained from external research to achieve the same goal.

It is necessary to highlight that both Engine A and Engine B are forks of an existing game engine written by myself, called the Topaz engine. This project claims technical contribution for Topaz in addition to Engines A and B, as the engine architecture being used is all from Topaz, a game engine that I have already written largely from scratch.

[5] The Unity Manual for the current most up-to-date Unity version (2018.2) can be found here: *https://docs.unity3d.com/Manual/index.html*

Engine A Development Process

The technologies constituting Engine A are wholly dependent on the outcome of the Preliminary Unity Research. The algorithms identified in the research that are used by Unity must be re-implemented in Engine A. This is done because these algorithms shall be compared with the algorithms eventually implemented in Engine B during the Algorithmic Analysis & Comparison process. Despite this, it is not necessary for the underlying architecture of Engine A to be similar to Unity; only the algorithms used. However, Engine A must follow the following guidelines:

- Engine A should have a simple and coherent architecture which the algorithms build upon. This is necessary because the underlying architecture of Engine B must be exactly identical to that of Engine A, and a simpler architecture shall better facilitate the differing algorithms used in Engine B. It is not necessary for Engine A to mimic the complex architecture of Unity, as the algorithms analysed are not strongly-coupled to the underlying architecture.
- Features of the engine must be as loosely-coupled as practically possible.

Engine B Development Process

The technologies constituting Engine B are wholly dependent on the outcome of the External Algorithmic Research. These algorithms will have been chosen with a specific corresponding Engine A/Unity algorithm in mind. It is imperative that aside from these differences in algorithm implementations, Engine B remain identical to Engine A in all other respects. Thus, Engine B must strictly follow the following guidelines:

- Algorithms identified in the External Algorithmic Research must be present in this game engine, and replace exactly one existing algorithm in Engine A.
- Aside from the aforementioned replacement algorithms, Engine B must remain exactly identical to Engine A. This includes external factors such as asset data (meshes and textures) and profiling technology.

Algorithmic Analysis & Comparison

In the Preliminary Research, the three properties of game development algorithms identified were performance, accuracy and flexibility. These three properties must be analysed in as much detail as possible, so that comparisons and contrasts can be made between each algorithm present in Engine A and its corresponding replacement in Engine B. Any anomalous results for either of these properties shall not be discarded from results. This is because the prevalence of outliers in the results affects the conclusion drawn from the results, so removing outliers could lead to treating one algorithm different than the other in terms of analysis, which must not happen. The methods through which analysis shall be performed are as follows:

Performance

Performance analysis can be carried out quantitatively via a profiler. Both Engine A and Engine B contain a TimeProfiler class, which provides the following useful software analytics:

- Time taken for any given code fragment to run (ms)
    - This is useful for profiling an arbitrary code fragment, such as a specific algorithm runtime duration.
- Average time between each frame (ms)
    - From this, the average number of frames-per-second can be calculated (FPS)

To ensure the fairness of profiling results, all algorithms used in all engines must be run on the same device using the same operating system under the same conditions, such as minimal and equal processes running in the background

Accuracy

All of the algorithms gathered from Preliminary Research and External Algorithmic Research (occlusion culling algorithms) have the following set of output cases for a function returning true if a given object is occluded in the camera's view:

- Wrongly returns true (The object is labelled as occluded when it should actually be visible). This prevents the algorithm from being conservative and will have a very noticeable result on the outputs rendered, as whole objects will disappear from the scene.
- Wrongly returns false (The object is labelled as visible when it is truly occluded). This leads to unnecessary GPU rendering of an invisible object. The principle of conservatism is however, not lost. However, this has absolutely no effect on the outputs rendered, as the erroneously-drawn object will be overdrawn by another.
- Correctly returns true (The object is successfully labelled as occluded and thus not drawn).
- Correctly returns false (The object is successfully labelled as visible and thus is drawn).

These cases show that there exists no output case where a small number of pixels are erroneous; either whole objects randomly disappear from view or the inaccuracy is completely invisible. For this reason, it is justified that the visible error-case can be verified by inspection instead of resulting to a computer-vision-based methodology of analysing frame-buffers.

This however, cannot be a valid methodology in the case where the inaccuracy is completely invisible (when an object is wrongly labelled as visible). For this reason, both Engines A and B contain a feature known as 'wireframe-mode'. When enabled, only the wireframes of the objects are visible; not the entirety of their faces. When enabled, objects which would normally be occluded by others shall remain visible. When occlusion culling wrongly labels an object as visible, this object will be clear when wireframe-mode is enabled. Thus, inspection remains a valid option for identifying these erroneous cases as long as wireframe-mode is enabled.

Flexibility

Flexibility of algorithms can be deduced via testing them under scenes resembling various game genres and environments. For a graphics-based algorithm, a diverse set of different scenes can be used to decipher whether the algorithm is flexible enough to function for different game application genres. Examples of these scenes include a closed-off maze-like scene to resemble games such as Wolfenstein 3D, or a large mountainous terrain scene.

Algorithms flexible enough to provide useful results in such a variety of game scenes would prove beneficial for a game engine. An excerpt from *Game Engine Architecture, Third Edition* states the following:

> "However, there is also a great deal of overlap--all 3D games, regardless of genre, require some form of low-level user input from the joypad, keyboard and/or mouse, some form of 3D mesh rendering..."(Gregory J., 2018)

The relevant section of the above quotation is in regards to 3D mesh rendering. The suggestion that this project makes is that although 3D mesh rendering should be universal in all 3D games and their engines, more complex concepts such as occlusion culling should also be universal in 3D games. It is for this reason that a flexible algorithm is required for these engines; that they may be able to support all games using the engine, with as few amendments to the algorithm required to any given game.

The algorithms identified in both the Unity research and the External Algorithmic Research are among the set of features expected to be available in games regardless of genre. For this reason, it is essential that the test scenes sufficiently represent the wide variety of game genres outlined previously.

Design

Engine Architecture

It may not be currently clear as to why Engine A is necessary, when Unity is already a game engine which implements its own algorithms. It is however necessary to re-implement the chosen Unity algorithm(s) in a new engine (Engine A). This is because of the following reasons:

- The underlying architecture of Unity is irrelevant to the properties of the algorithms used to achieve certain game engine functionality
- Unity is closed-source and is technologically very different to Engine B, thus cannot be profiled in the same manner and fairness of comparisons cannot be guaranteed.

The algorithm used in Unity cannot be changed. This means that amending the conditions where a valid comparison could be made between Unity and Engine B is impossible. If the chosen algorithms are developed in a separate engine (Engine A), the underlying architecture of that engine can be identical to the one used in Engine B. This will aid in analysis of the algorithms in regards to performance, as profiling results will vary based only on differences in algorithm implementations, not on differences of architecture of the engines.

Scene Architecture

Scenes are collections of objects representing a given game world. Performance, accuracy and flexibility metrics will be gathered for both occlusion culling algorithms working on the following three types of scenes:

- Static Maze Scenes
- Dynamic Maze Scenes
- Static Terrain Scenes

Static scenes are scenes which contain no moving objects. Objects are considered moving if any of their position, rotation, or scale changes during runtime. Dynamic scenes are scenes which contain at least one moving object.

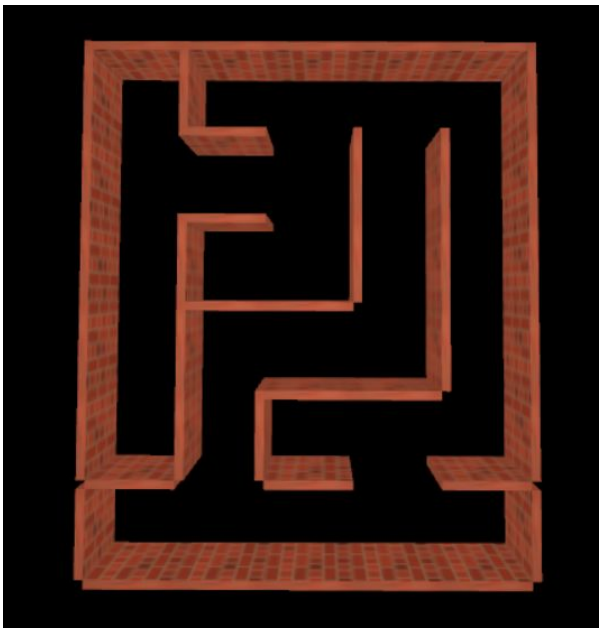
These scene types were chosen in an effort to ensure that the scenes ran could be expected to be found in a typical 3D game. If the scenes chosen are diverse in their layout and density, then metrics gathered with respect to those scenes are not too niché; they can approach the 'average scene' which may be expected in a 3D game.

It is, however, not sufficient to merely have one scene of each type. Not all static mazes are the same, for example. There may be mazes which are very complex, or large in scale compared to other scenes. In order to closely approach the general case, it is necessary to ensure that multiple scenes of the same type exist with varying parameters. For example, gathering metrics from a large, complex scene in addition to from a small, simple scene will show how these changes in parameter affect the metrics. From there, it can be identified whether a change in metrics is because of the scene type, or simply the architecture of that specific scene.

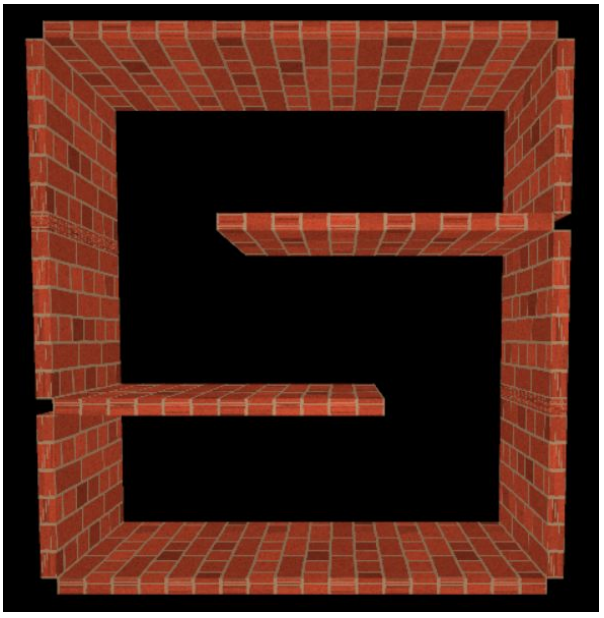
For this reason, there shall be three scenes for each type, each with varying parameters. For example, for both static and dynamic scenes, each of these three scenes shall contain one of the following:

- A medium-complexity, medium-scale maze, known as Maze 1.
- A low-complexity, large-scale maze, known as Maze 2.
- A high-complexity, small-scale maze, known as Maze 3.

The maze designs for Maze 1, 2, and 3 are visible below.



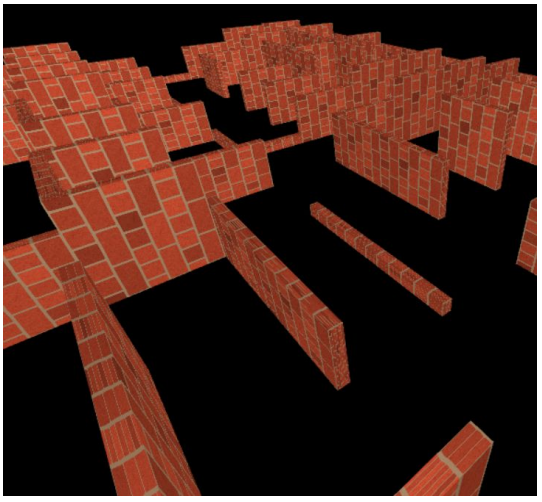
Pictured: A birds-eye-view of Maze 1



Pictured: A birds-eye-view of Maze 2

Pictured: A birds-eye-view of Maze 3

These are the static versions of the maze scenes. The three dynamic scenes shall be identical to these three scenes aside from one property: The heights of the walls of the mazes shall vary with respect to time in a sinusoidal manner. Below is an example of a time instant in Dynamic Maze 3 where the heights of the walls vary:



Quantifying Performance

As stated previously, an essential relationship between Engines A and B is that they are completely identical aside from their methods of performing occlusion culling. If two largely identical benchmark programs linking to engines A and B respectively are using the same compiler & settings in addition to the same hardware, profiling the number of milliseconds per frame taken up by both occlusion culling algorithms result in a fair comparison between the two algorithm implementations.

Thus, the performance metrics gathered for each scene are as follows:

- Mean execution time per frame
- Standard deviation of execution time per frame

These metrics are in regards to the runtime of both occlusion culling algorithms only in every frame. This does not include rendering time. The reason that rendering time is not included is because the accuracy of the algorithms shall also affect the rendering time. It is necessary that the performance characteristic is completely uncoupled from the accuracy characteristic.

A requirement of the implementation of the occlusion culling algorithm used in Engine A (called PVSOC) is that the current scene must be split into a set of nodes. Each node has a midpoint. Sampling the performance in one location in the scene is not sufficient to gather enough data to quantify performance. Instead, the performance metric shall be gathered in the midpoint of each node. As larger scenes require a greater number of nodes, the number of performance metrics gathered is equal to the number of nodes in the scene. This means that larger scenes will have more performance metrics gathered to ensure that the vastness of the scene is adequately visited.

In Engine B where there is no PVSOC, there are no nodes. For this reason, the node midpoints shall be hardcoded into Engine B so that samples can be extracted from the exact same camera position and orientation to ensure that the comparison remains fair. The memory and time overhead incurred from reading from these hardcoded orientations is assumed to be negligible.

The performance of an algorithm is dependent on two factors related to the performance metric: The mean length of time necessary to execute the algorithm, and the consistency of such durations. The performance metrics of both engine demonstration programs can be plotted on one graph to allow the results to be easier to visualise. To easily quantify the consistency of the performance metrics, a standard deviation can be calculated for a given scene, to easily compare the consistencies of both algorithms.

Individual performance metrics are gathered more frequently than any others. Individual performance metrics are gathered X times, per node, per scene, per engine. However, the metrics presented shall include a graphical representation of the individual metrics aswell as the total metrics per scene, per engine. This simplifies the analytical process without losing information, as any outliers shall be easily visible in the graphical representation of the data. Thus, there shall be one set of performance metrics per scene, per engine.

Quantifying Accuracy

occlusion culling algorithms should expect an input corresponding to some 3D object belonging in a scene. The output of the algorithm is a simple boolean; true if the object is occluded, and false if it is not. Thus, the accuracy metrics are as follows:

- Number of objects wrongly labelled as visible.
- Number of objects wrongly labelled as occluded.

It is important to note that these metrics do not take frustum-culling into account. This means that if objects outside of the camera's view frustum are wrongly-labelled as visible, then this will not count towards the metric as the metrics assume that frustum-culling has already removed these objects from the rendering pipeline.

An essential property of an occlusion culling algorithm is that it is conservative. An excellent description of the conservative and aggressive property is provided in the paper 'Exact From-Region Visibility Culling':

> "Conservative techniques consistently overestimate visibility and incur a false visibility error: invisible polygons are considered visible. This results in sub-optimal run-time performance because these polygons are unnecessarily submitted to the rendering pipeline. In contrast, aggressive methods always underestimate the set of visible geometry

> and exhibit false invisibility, where visible polygons are erroneously excluded. Aggressive visibility causes image error but can be useful in practice if: (a) the perceptual impact of the error is acceptably small, (b) the algorithm is computationally efficient or (c) it handles scenes that cannot be solved effectively with a conservative alternative due to excessive overestimation" (Nirenstein S., Blake E., Gain J., 2002)

In the case of rendering scenes into a window, an aggressive technique is unacceptable: (a) is not applicable because it can never be guaranteed that the impact of the error is small, (b) is not applicable because in this specific project, performance is not the only criterium, and (c) is not applicable because there do exist conservative alternative techniques. It is for this reason that conservative techniques would be valued far more than aggressive techniques with respect to accuracy.

Thus, individual accuracy metrics are gathered once per node, per scene, per engine. The metrics to be presented will be a summation of each metrics for all nodes in a scene. This means that there shall be one set of accuracy metrics per scene, per engine. Accuracy could be better quantified if metrics are gathered for every single possible orientation and position within the scene. However, this would lead to an immeasurable quantity of samples, and thus gathering metrics once per node in each scene is a justifiable compromise.

Quantifying Flexibility

As stated previously, flexible algorithms are able to be applied to a diverse range of scenes with as little amendments required to allow the algorithm to work on all scenes properly. The research into this property has lead to the following scene samples used while gathering metrics:

- Three maze-scenes of varying complexity and physical scale
- Three dynamic maze-scenes otherwise identical to the initial maze-scenes, but with dynamic objects. Dynamic objects are objects where either the position, rotation or scale are not constant throughout the program runtime. An example of a dynamic object would be the height of a maze-wall varying per frame.
- One large, regular, terrain mesh scene. Examples of such scenes could be grassy landscapes, desert biomes or jungles.

The specific flexibility metrics gathered are as follows:

- Number of general scene preprocessing steps
    - This is equal to the number of amendments required to accommodate a specific scene. This can be applied to either the algorithm implementation or the external file containing the scene data.
- Number of object-specific preprocessing steps
    - This is equal to the number of processing steps that need to be performed on each object before runtime.
- Number of additional metadata per object
    - This is equal to the number of extra pieces of data required by each object to assure that its occlusion culling algorithm can run properly.

One set of flexibility metrics shall be gathered exactly once per scene, per engine.

Implementation

As a result of the Unity Research and External Algorithmic Research, the specific technologies being investigated in this project are all related to the topic of occlusion culling. Occlusion culling is the process of identifying whether objects in the scene are visible to the camera, and thus should be rendered. The purpose of occlusion culling is for optimisation; if objects can be labelled as invisible before the scene is rendered, invisible objects can be discarded from the render pipeline, improving performance.

Hardware & Platform Specifications

- Engine A and Engine B are compiled under C++17 via g++, using the gcc version 7.1.0 (x86_64-posix-seh-rev2) built by the MinGW-W64 project. Build scripts are provided via CMake. The minimum version of CMake required to build the project is CMake version 3.9.
- The version of Unity used for the project is 2018.2.15. The Scripting Runtime Version used is .NET 3.5 Equivalent, with the backend powered by Mono.
- The version of OpenGL used is 4.6
- External Libraries Utilised in Engine A and Engine B:
    - GLEW
    - Assimp
    - SDL2 (including addons)
        - SDL2_mixer
        - SDL2_ttf
- The implementation of occlusion culling in Engine A, based upon potentially-visible-sets is a custom implementation created by myself. This is not the case for the occlusion culling implementation of Engine B, this is from an external library that was written by a third-party.
- External Libraries used exclusively in Engine B:
    - Masked Software Occlusion Culling

To ensure validity of profiling results, all three engines used (Unity, Engine A and Engine B) shall run on the following technology:

- Processor: Intel Core i5-3570k CPU @ 3.40GHz, 3401Mhz, 4 Cores, 4 Logical Processors
- Graphics Card: NVIDIA GeForce GTX 660
- Physical Memory (RAM): 8.00GB
- Operating System Name: Microsoft Windows 10 Home (x64-based PC)
- Operating System Version: 10.0.17134 Built 17134

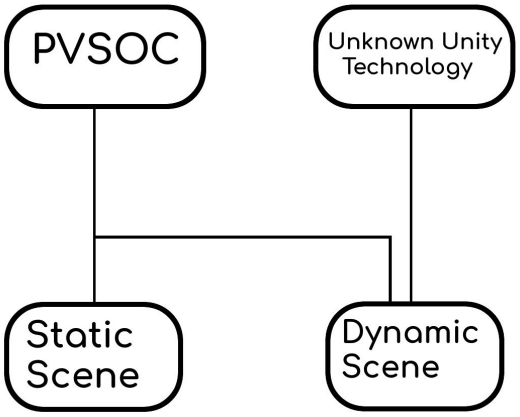
Currently, all algorithms outlined in this implementation are a method of achieving occlusion culling. Without occlusion culling, a 3D graphics application will draw all objects in a scene that were marked to be rendered. This includes objects which are not visible from the camera because their view is blocked (occluded) by another object. As stated previously, occlusion culling is the act of identifying these occluded objects and discarding them from the rendering process; thus saving rendering time and improving performance.

Unity & Engine A Algorithms

The following algorithm was discovered during the Unity research process:

Potentially Visible Set Occlusion Culling (PVSOC)

PVSOC is a 3D graphics algorithm consisting of a preprocessed scene and light CPU runtime to achieve occlusion culling. This is used by both Unity and Engine A. Unity uses PVSOC for both static and dynamic scenes. This is surprising because PVSOC typically only works for static scenes. Static scenes contain no moving objects whereas dynamic scenes contain at least one moving object. This means that Unity is utilising an additional technology aside PVSOC in dynamic scenes. This is evident in the following diagram:



As Unity is closed-source, there is no way to feasibly replicate this unknown technology in Engine A. For this reason, when profiling this technology, it is expected that PVSOC may perform poorly on any dynamic scenes, as this would not be utilising the unknown additional component in Unity.. In Unity's manual describing its occlusion culling, the explanation has continued to refer to PVSOC since Legacy Documentation version 5.0.[6] Unity 5 was released on the 3rd of March, 2015, meaning that Unity has used PVSOC for several years now, as of April 2019.

A major disadvantage of PVSOC is its requirement of scene preprocessing. This is because the quality of the occlusion culling provided by PVSOC is dependent on multiple factors:

- Organisation of objects in the scene
- Efficiency of the preprocessing algorithm
- Whether the scene is dynamic or static

The organisation factor is particularly troublesome; the quality of the algorithm varies massively depending on how the scene is structured. For this reason, the external algorithms chosen as replacements in Engine B were chosen specifically for their superior flexibility.

Engine B Algorithms

The following algorithms were discovered during the external algorithmic research process:

Masked Software Occlusion Culling (MSOC)

MSOC is a 3D graphics algorithm consisting of a software-rendered depth-buffer heavily utilising the CPU to achieve occlusion culling. It is used by Engine B. MSOC is the subject of the paper 'Masked Software Occlusion Culling', Hasselgren et. al. (2016) of the Intel corporation, whom also innovated the algorithm. MSOC is an improvement to an older Intel algorithm simply named Software Occlusion Culling (SOC). SOC was, however, determined to be too resource-intensive to be appropriate for a game application.

Both MSOC and SOC achieve occlusion culling by doing the following:

- Software rendering a custom depth-buffer for the scene in RAM. Software rendering is different to typical graphics processing because it happens on the CPU and does not have the advantage of being hardware-accelerated by the GPU.

[6] Unity Legacy Documentation version 5.0:
https://docs.unity3d.com/500/Documentation/Manual/OcclusionCulling.html

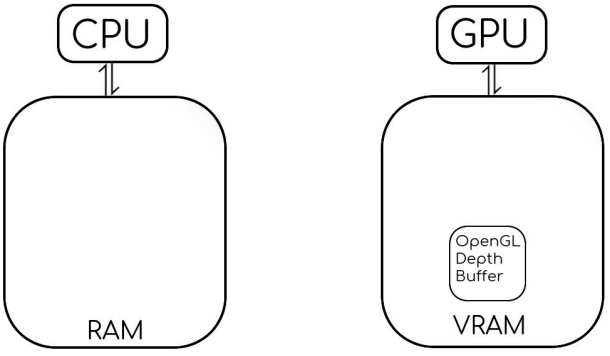
- This custom depth-buffer is used only by the CPU to perform custom depth-testing before any draw-calls are sent to the GPU. This is very different to the typical practise of issuing a draw-call to the GPU, where the depth-test is then performed using a depth-buffer stored in VRAM.
- Based upon these depth-tests, the CPU can issue draw-calls to the GPU for *only* objects that are not occluded and thus need to be rendered. Occluded objects which are not visible to the camera do not issue corresponding draw-calls to the GPU at all. This achieves occlusion culling.

For each object in the scene, the RAM depth-buffer is used to perform a depth-test to deduce whether the object is occluded. If the object is deduced to be occluded, then a draw-call is avoided and the whole duration of rendering that object via the GPU is avoided.

There is, however, a major issue with this SOC approach: Consider a typical modern framebuffer of resolution 1920x1080. Consider also a size of four bytes for a single floating-point value. The size of a 1080p depth-buffer for this display would necessitate $1920 * 1080 * 4 = 8,294,400\ bytes$. This means ~8MB is to be processed every single frame continuously. In isolation, it could be possible to perform this work in a rate approaching 60FPS, but not for a game environment. Running at 60FPS would mean that an average frame would require *at most* ~16.6ms of processing. Even simply iterating through an 8MB depth-buffer 60 times a second would require *at most* $\frac{1}{60 * 1920 * 1080 * 4} = 7.5352045e-10s$, i.e ~75ns of CPU time spent processing per iteration. Performing a depth-test within such a small period of time would almost certainly be insurmountable for a typical CPU, especially in an environment such as a game which may be already demanding on CPU utilisation.

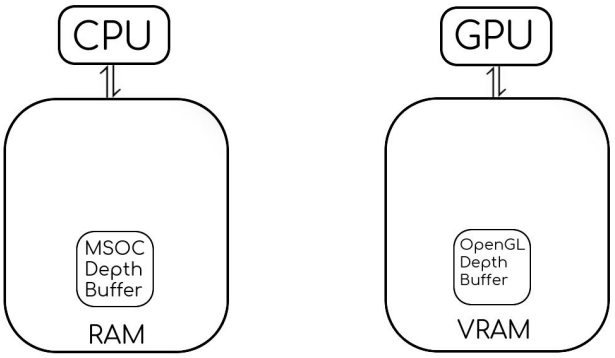
MSOC utilises a common lossless bitmap-compression technique to drastically reduce the size of the depth-buffer without losing any detail. This is done by considering the depth-buffer as a bitmap. Each pixel value corresponds to a float already existing in memory, effectively preventing duplicate depth values in the depth-buffer. This results in an otherwise ~8MB depth-buffer being reduced drastically enough to have acceptable performance.

The GPU is a piece of hardware specifically designed to excel in performing depth-tests, and indeed is far superior in doing so over software-rendering the depth buffer. For this reason, it seems illogical to push such work onto the CPU when the GPU is far superior for carrying out this work. However, by performing all processing on the CPU, no occlusion data needs to be sent to the GPU at all; avoiding the CPU-GPU latency cost entirely. To aid the point, consider the following diagram:



As standard in typical computer architecture, the CPU and RAM are connected by a bidirectional bus. So too are the GPU and VRAM on the graphics card (assuming we are using a discrete graphics processor as opposed to an integrated one). To perform occlusion culling, we must have access to the depth-buffer; this block of memory contains all the depth information about every single object in the scene; ultimately, which fragments are occluded by others. Although the CPU can dispatch commands to the GPU to indirectly communicate with the depth-buffer, this cannot

be done without a significant performance penalty. The diagram below very simply illustrates the solution that MSOC proposes:



As can be seen, there is an additional depth-buffer in RAM which the CPU can easily use to perform occlusion culling without aid from the graphics card at all. This also allows for some or all of the MSOC depth-buffer to be stored in the CPU cache, which could drastically reduce the performance penalty incurred from software-rendering a depth-buffer.

MSOC could be used instead of PVSOC because of the following benefits:

- Unlike PVSOC, MSOC requires no preprocessing of the scene. This may lead to improved flexibility as scene designers shall not need to optimise their layouts to yield better preprocessing results. This would be necessary because preprocessing different scenes yields varying performance results based upon the organisation and densities of objects in the scene.
- Both algorithms run on the CPU, so cannot be hardware-accelerated by the GPU. However, as MSOC involves building a depth-buffer in RAM, it receives a more substantial benefit from CPU parallelisation; specifically from multithreading and utilisation of modern CPU SIMD functionality.

In addition, working on the CPU allows the use of many different optimisation techniques unique to the CPU. An example of this is vector extensions on Intel CPUs (Reinders J., 2013). As SIMD support increases, the effectiveness of software-rendering a depth-buffer becomes vastly improved, as the number of instructions which need be executed in order to render to the depth-buffer drops massively. This holds especially true if 512-bit SIMD support becomes the norm, as opposed to more conventional SIMD support such as 128-bit or 256-bit. Such optimisations would not be available to a GPU-based approach, and would be more applicable in MSOC than PVSOC, as the task of rendering a depth-buffer in MSOC can benefit more from vectorisation than tasks in PVSOC. The same holds true for parallelisation; multithreading is extensively mentioned in Masked Software Occlusion Culling, and (in addition to SIMD) is among the exhaustive measures taken to allow the CPU-approach to be realistically manageable (Hasselgren J., Andersson M., Akenine-Möller T., 2016).

It is possible that MSOC will occupy the majority of the modern CPU cache in addition to CPU utilisation and yield a poorer performance than the more lightweight PVSOC. If this were the case, MSOC would be considered superior only in the case where the loss of performance could be justified by the increase in flexibility and accuracy.

OpenGL Query Objects (GLQO)

GLQOs are hardware-accelerated constructs created by the CPU to be dispatched to VRAM, where the GPU processes the request until a result is available upon request by the CPU. GLQOs are used by Engine B. They are supported natively by OpenGL since version 1.5.[7] The algorithm utilising GLQOs works as follows:

[7] OpenGL documentation on GL Query Objects for Occlusion Culling:
https://www.khronos.org/opengl/wiki/Query_Object#Occlusion_queries

1. GLQO created by the CPU in RAM.
2. The query command is dispatched to the GPU.
3. Occlusion meshes are then dispatched to the GPU via a draw-call.
4. The GPU processes the draw call and updates the query as to whether the occlusion query passed or not. Meanwhile, the CPU is waiting for the GPU to finish processing and should be performing other work.
5. Once the GPU finishes processing the query object, it waits for additional draw-calls. Meanwhile, the CPU obtains the result of the draw-call, and decides whether to render the given mesh or not.
6. If the mesh should be drawn, then finally the GPU stops being idle, receives the draw-call and the mesh is rendered. Otherwise, nothing is drawn.

My research yielded that GLQOs could be used instead of PVSOC for the following reasons:

- Like MSOC, the other possible replacement, GLQO usage requires no scene preprocessing at all. This is advantageous because composition of a scene (done by game designers, not developers) has less impact on the performance and allows for more creative and interesting scenes; thus increasing flexibility.
- PVSOC runs its occlusion checks entirely on the CPU. With GLQOs, the workload is instead dispatched to the GPU; a hardware component specialised perfectly to execute occlusion checks by depth-testing. This can reduce the likelihood of CPU throttling from overuse and consuming much of the CPU cache, allowing it to be used by other parts of the game program to improve performance.

There is a subtle, yet crippling issue with this methodology which poses a major drawback. The issue can be described by the following excerpt from '*Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization*':

> "However, GPU-based acceleration is no silver bullet, and offloading to the GPU is not free. In both discrete and integrated GPUs, there are relatively large overheads associated with data transfer, kernel launch, and synchronization." (Lustig D., Martonosi M., 2013)

Lustig D., Martonosi M. proposed also a solution to reducing this latency somewhat. However, like all proposed solutions to this issue, none achieve a major reduction in the delays between components. Unfortunately, there is no known solution to greatly reducing this latency. This latency all-but nullifies the time saved by utilising hardware acceleration in the first place.

The algorithmic research indicates that the lack of accuracy incurred by the aforementioned issue means that without amendments, the algorithm shall not be conservative. As stated previously, any algorithm that is not conservative is very unlikely to be a justifiable replacement for PVSOC under these circumstances.

In addition, there is a time-constraint component to GLQOs. Both PVSOC and MSOC are CPU-based approaches, thus the methodology of algorithmic analysis and comparison need only cover the CPU-side, not the GPU-side. GLQOs utilise hardware acceleration (GPU). Thus, to adequately analyse GLQOs, it would be necessary to amend the algorithmic analysis to ensure that all GPU-side attributes (such as GPU time incurred by GLQO overhead) are measured. Combined with the unlikely case of it being a justifiable replacement for PVSOC, GLQOs were ultimately not added to Engine B and shall not be included in the algorithmic analysis.

Evaluation

Static Maze Scenes

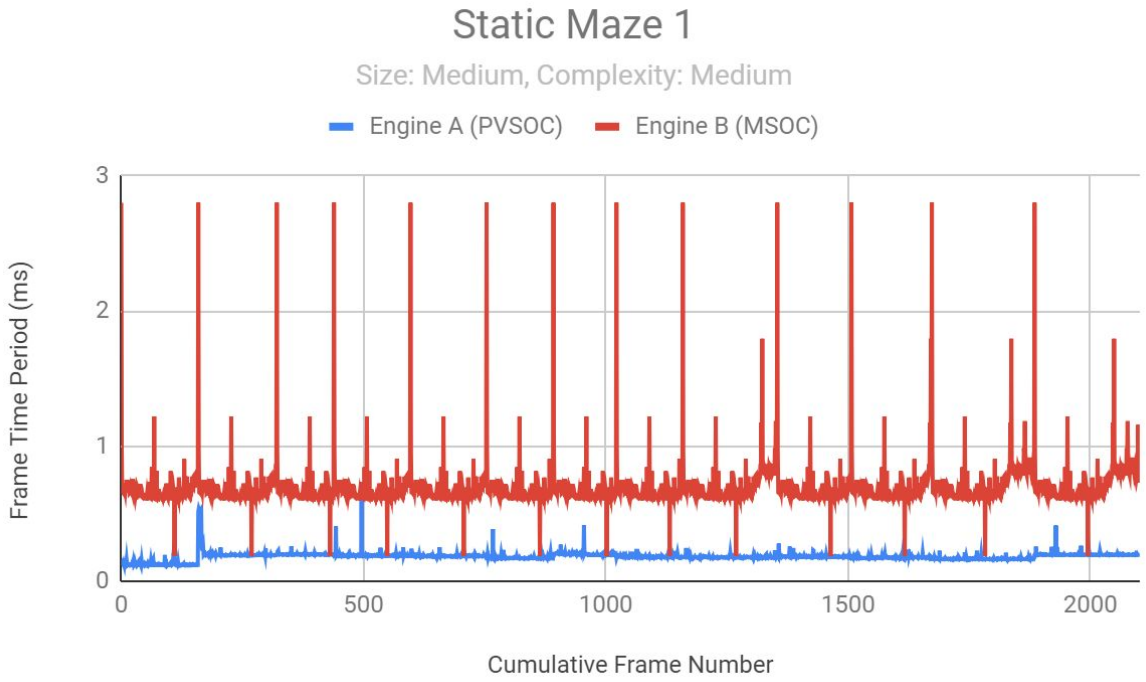
Static Maze 1

Performance

The gathered metrics for Static Maze 1 are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.182	0.692
Standard deviation of execution time per frame (ms)	0.0319	0.199

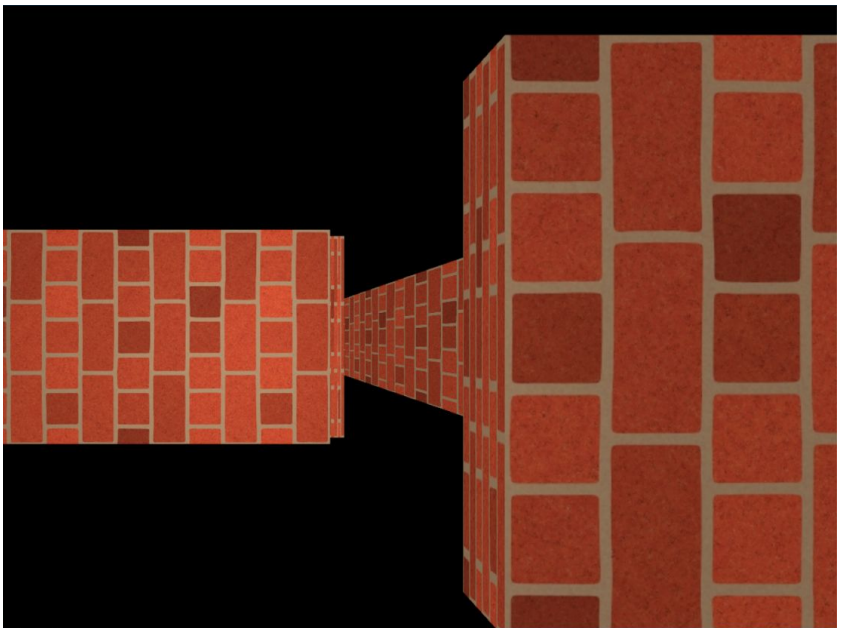
All data samples are plotted graphically below.



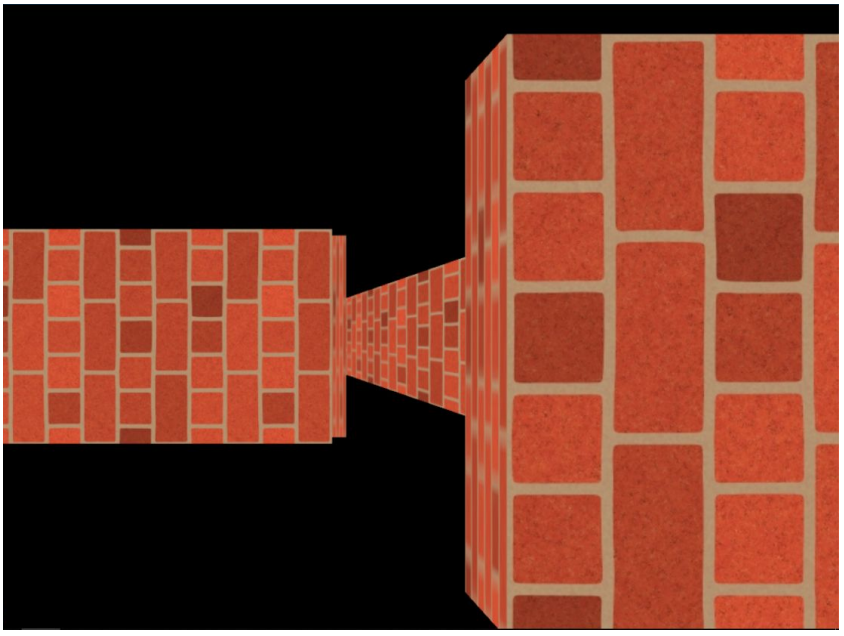
The mean execution time of PVSOC per frame is 0.182ms, under a third of the mean execution time of MSOC. This means that in the average case for this specific scene, PVSOC clearly runs faster than MSOC. In addition, the standard deviation for PVSOC is an incredibly low 0.0319ms, implying that the performance is very consistent. The chart also illustrates that the blue line representing PVSOC is far more consistent than a more volatile MSOC. This is strong evidence that PVSOC consistently runs much faster than MSOC for this medium-complexity medium-size maze scene.

Accuracy

Below is a screenshot of node B in Maze 1 in Engine A.

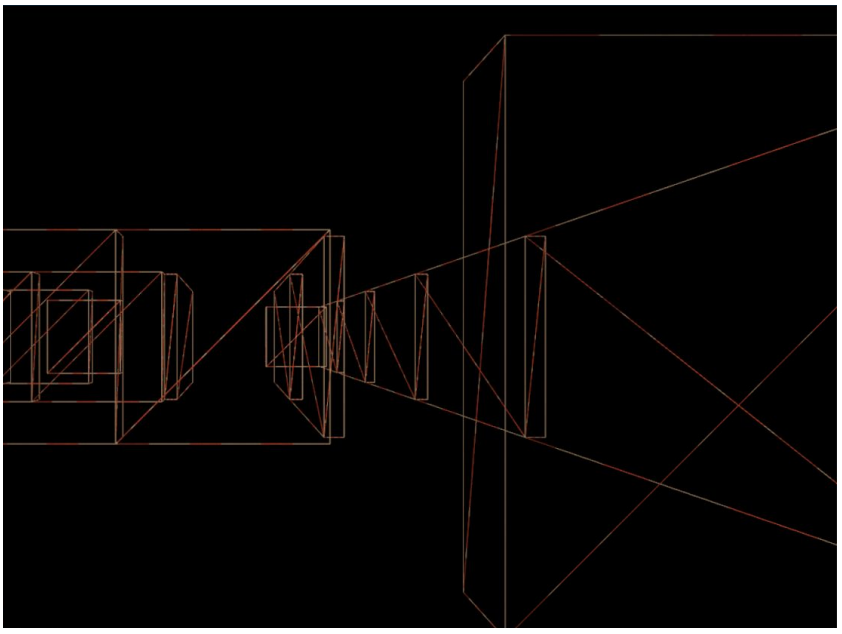


This framebuffer can be compared with a corresponding screenshot of node B in Maze 1 to see if there are any differences:

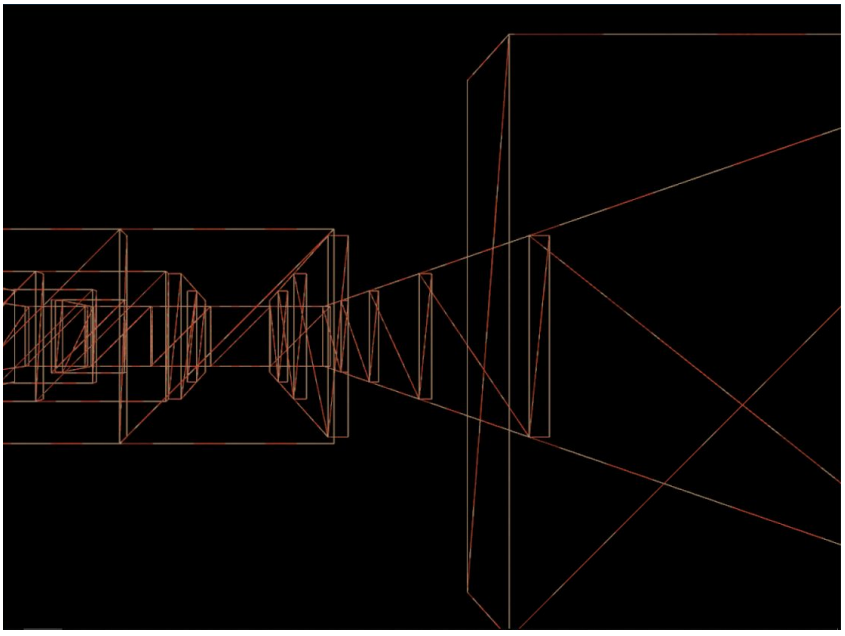


It is clear that the images are identical. This means that in node B of Maze 1, PVSOC has not wrongly labelled any objects as occluded. If this is the case for every node in Maze 1, this means that the number of objects wrongly labelled as occluded will be zero; meaning that this implementation of PVSOC is conservative.

This is the same node B using PVSOC but in wireframe mode.



It is already clear that several objects have wrongly been labelled as visible. However, the same inspection methodology used for the other accuracy metric should be used again for this metric in wireframe-mode. Below is the same node B in wireframe mode, but with occlusion culling disabled:



Although PVSOC has clearly succeeded in preventing several invisible objects from being rendered, multiple of these invisible objects have not been prevented from being rendered. The number of objects wrongly labelled as visible in this node is seven. These discrepancies can be identified in the above images; there are many objects that are being rendered despite being invisible in the screenshot without occlusion culling, but there are seven objects in total whose wireframes are visible in the PVSOC screenshot despite clearly not being visible to the camera outside of wireframe-mode.

The same process is repeated for every node in Maze 1, and again for Engine B. The metrics below represent the total gathered metrics for every node in Maze 1 for Engine A and Engine B.

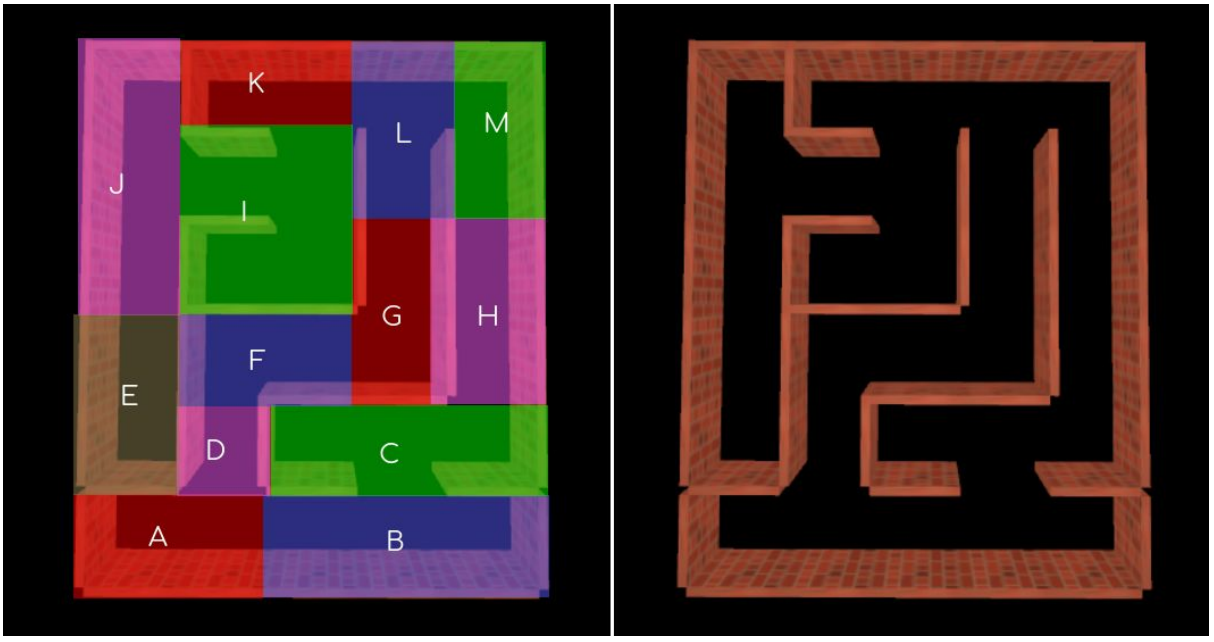
	PVSOC	MSOC

Number of objects wrongly labelled as visible	72	14
Number of objects wrongly labelled as occluded (Zero means conservative)	0	0

It is clear from the table that both implementations remain conservative. This is good; an occlusion culling algorithm that is not conservative has disastrous consequences for the end-user. However, this implementation of PVSOC has erroneously labelled many more objects visible than MSOC did. It is very clear that MSOC was able to prevent far more invisible objects from being rendered than PVSOC for this specific scene. Whether the lost time in occlusion culling processing is regained by the reduced rendering time is dependent on the specific scene. In this case, it was not; PVSOC yielded better overall program performance in addition to just the execution of the algorithm mentioned in the performance metrics.

Flexibility

The scene nodes of Maze 1 are visible here:



Left = Scene with node labels, Right = Scene without node labels

There are thirteen nodes total. Thus, the gathered metrics for Static Maze 1 are as follows:

	PVSOC	MSOC
Number of general scene preprocessing steps	13	0
Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

The table suggests that MSOC requires no preprocessing nor additional element of metadata per object in the scene. This is true. MSOC simply builds a hierarchical depth-buffer in RAM and builds it as occlusion queries are produced. This means that all MSOC processing is done in real-time. This may be partially why its performance metrics were frequently worse than PVSOC, although it has the advantage of making MSOC incredibly flexible.

This specific scene required thirteen nodes to partition the objects into to allow PVSOC to function. Each of these are discrete scene preprocessing steps, as the bounding volumes representing the regions had to be constructed based upon their object dimensions. MSOC requires no such preprocessing.

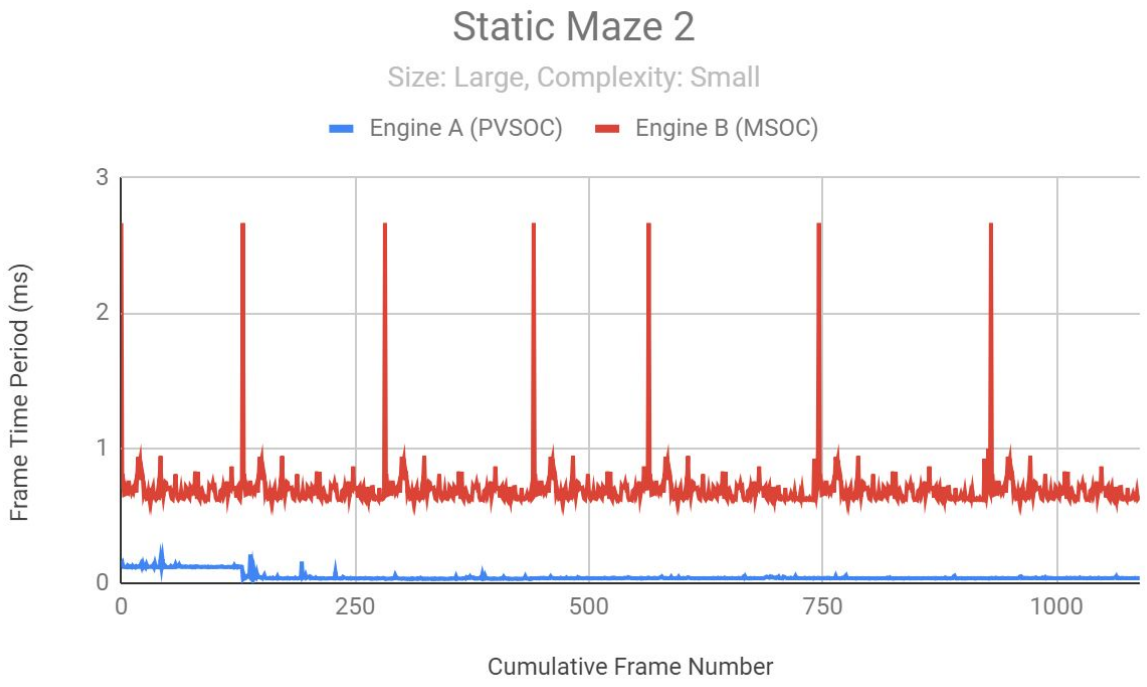
Static Maze 2

Performance

The gathered metrics for Static Maze 2 are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.0478	0.696
Standard deviation of execution time per frame (ms)	0.0291	0.171

All data samples are plotted graphically below.



Each object comprising Static Maze 2 is exactly ten times greater in scale than the objects in Static Maze 1. However, Static Maze 2 is a much simpler maze comprised of fewer objects than Static Maze 1. The mean execution time for PVSOC is consistently far below that of MSOC. It is clear from the chart, mean and standard deviation calculations, that PVSOC consistently outperformed MSOC in this scene.

The performance metrics for MSOC are virtually identical in this scene than the previous, despite the parameters of the scene being different. This suggests that there is a large inherent overhead attributed to MSOC, but not much at all in PVSOC. For larger scenes, it may be possible that the inherent overhead attributed to MSOC becomes insignificant compared to the growth-rate of the number of objects in the scenes. Although there is currently not sufficient evidence to back up this idea.

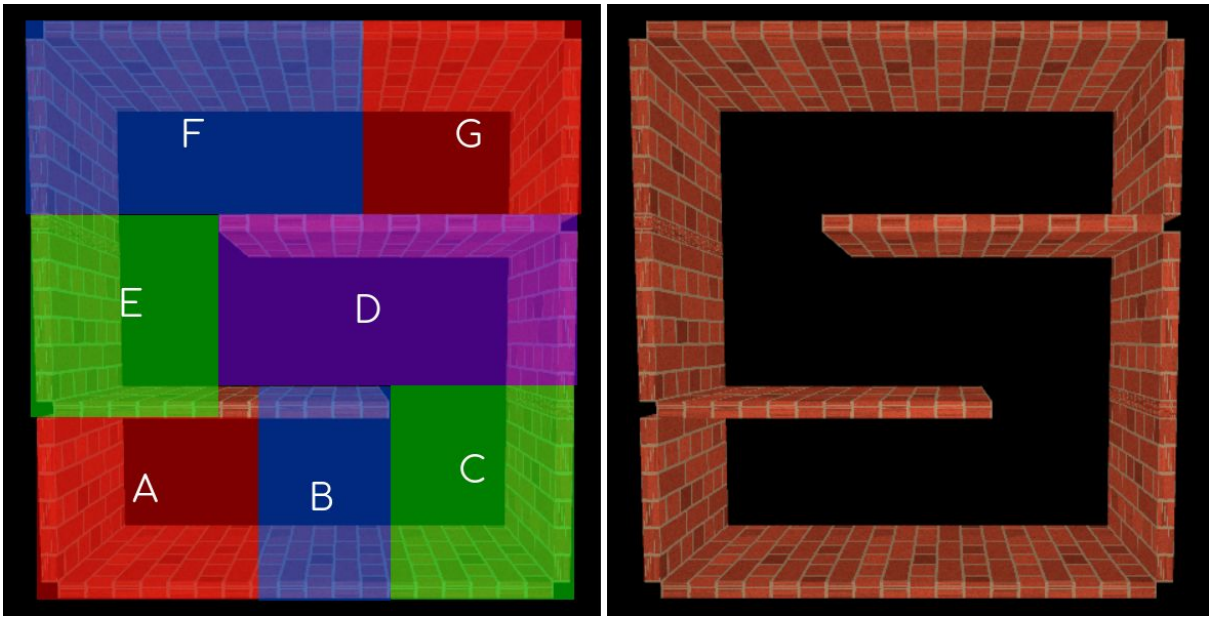
Accuracy

	PVSOC	MSOC
Number of objects wrongly labelled as visible	19	16
Number of objects wrongly labelled as occluded (Zero means conservative)	0	0

It is clear from the table that both implementations remain conservative. In this much simpler scene, PVSOC was able to avoid wrongly marking as many objects as visible, although still not quite to the degree of MSOC. This could suggest that PVSOC produces more accurate results on simpler scenes, although there is not yet sufficient evidence to prove this. Like Static Maze 1, MSOC has proven that it can provide more accurate results than PVSOC, although to a less extreme degree in this scene.

Flexibility

The scene nodes of Maze 2 are visible here:



Left = Scene with node labels, Right = Scene without node labels

There are seven nodes total. Thus, the gathered metrics are as follows:

	PVSOC	MSOC
Number of general scene preprocessing steps	7	0
Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

As before, MSOC required absolutely no additions nor preprocessing in order to accommodate this scene. PVSOC still needed to construct seven separate nodes to partition the scene nodes into, which each are considered scene preprocessing steps.

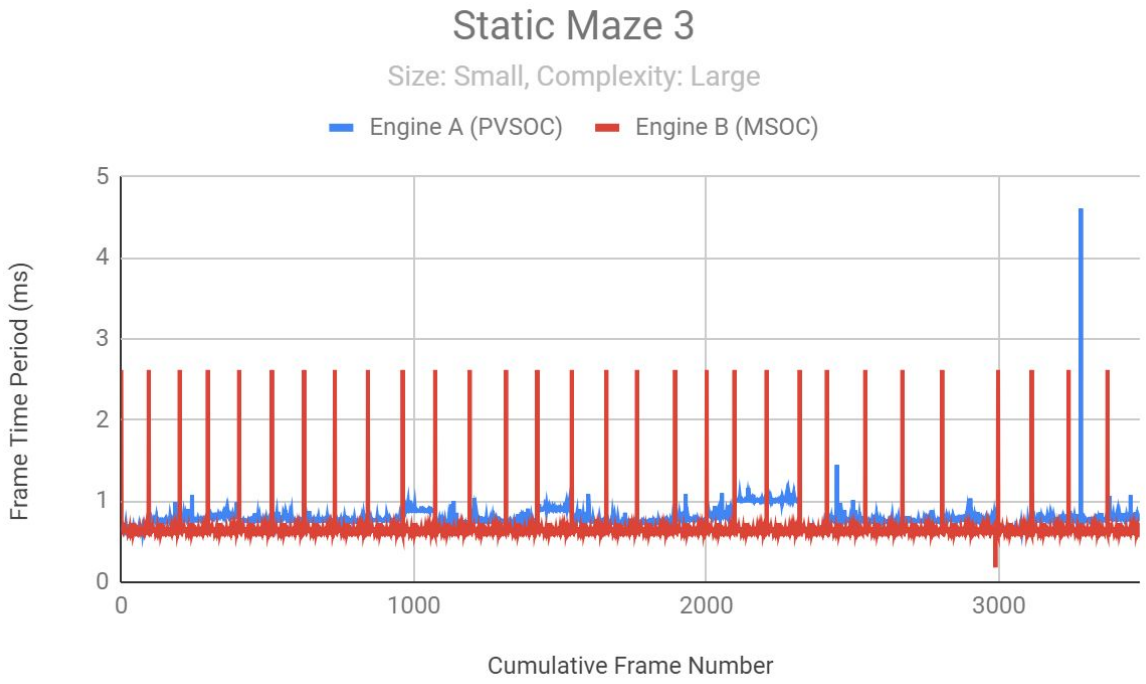
Static Maze 3

Performance

The gathered metrics for Static Maze 3 are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.787	0.650
Standard deviation of execution time per frame (ms)	0.107	0.185

All data samples are plotted graphically below.



Each object comprising Maze 3 is exactly one tenth of the scale of an object in Maze 1. The complexity & number of objects in this scene, however, is far larger than that of Mazes 1 and 2. Interestingly, the performance metrics do not suggest the same as they were earlier. In both Mazes 1 and 2, PVSOC consistently outperformed MSOC by a considerable margin. In this more complex maze, this is not the case. This time, PVSOC was consistently outperformed by MSOC, if the outlier samples in MSOC are to be discarded. As stated previously, however, the outlier samples were not discarded yet the mean execution time per frame was still lower.

The suggestion in the previous maze scene would be that if a scene had a very large number of objects, the growth-rate of the occlusion culling algorithms would overshadow the inherent overhead of MSOC. The performance metrics from this maze support that theory.

Accuracy

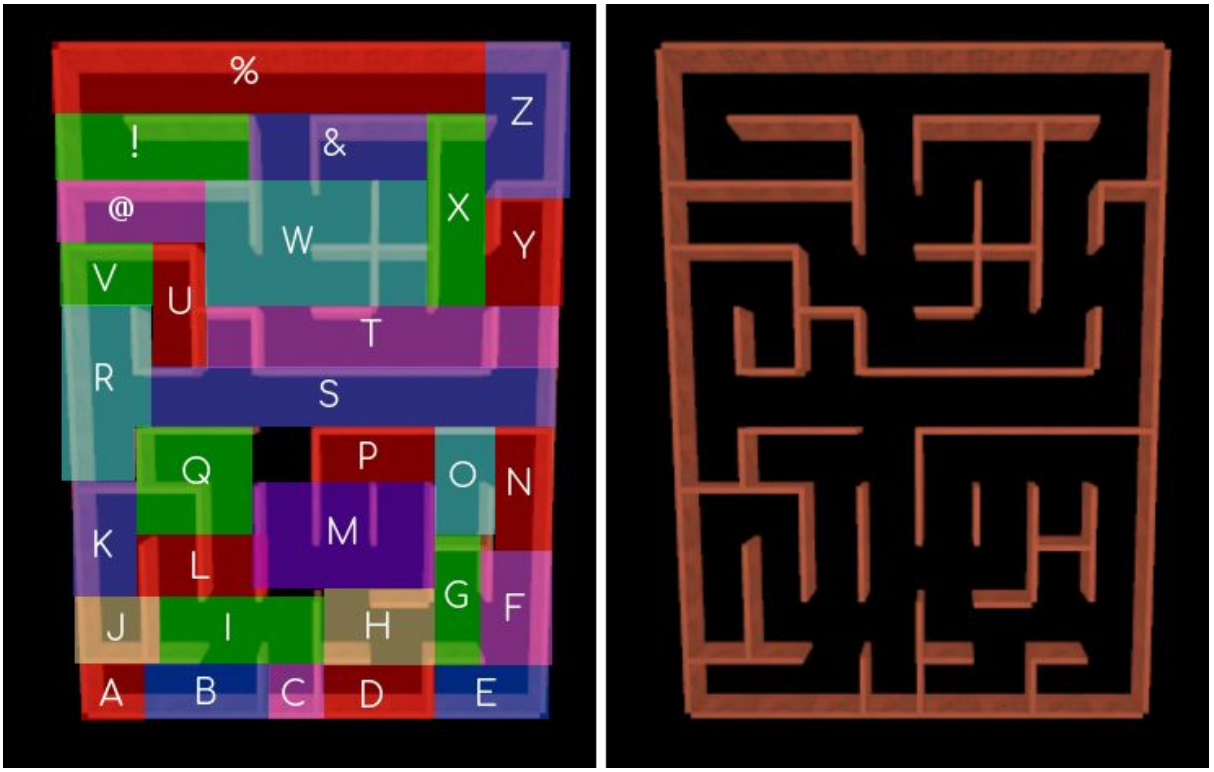
	PVSOC	MSOC

Number of objects wrongly labelled as visible	219	157
Number of objects wrongly labelled as occluded (Zero means conservative)	0	0

Once more, this data supports the notion that both of the algorithm implementations are conservative. A previous suggestion from Maze 2 was that PVSOC may provide more accurate results for simpler scenes. Although Scene 3 is a large scene with very many objects, 219 wrongly labelled objects is very large and thus this data does not support that idea. This data does, however, continue the trend of MSOC yielding more accurate results than PVSOC for static maze scenes.

Flexibility

The scene nodes of Maze 3 are visible here:



Left = Scene with node labels, Right = Scene without node labels

There are thirty nodes total. Thus, the gathered metrics are as follows:

	PVSOC	MSOC
Number of general scene preprocessing steps	30	0
Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

As this maze scene was much more complex than the others, a total of thirty nodes needed to be preprocessed for this scene. The architecture of MSOC allows it to continue to require no such amendments.

Static Maze Conclusion

The data from the varying parameters of each static maze scene make some intriguing suggestions:

In the simplest maze, Maze 2, PVSOC consistently outperformed MSOC by a large margin. In a slightly more complex maze, Maze 1, PVSOC still consistently outperformed MSOC but with a lesser margin. In a much more complex maze, Maze 3, the performance metrics suggest that both algorithms performed very similarly, although MSOC did consistently match or outperform PVSOC in performance, albeit by a small margin. From this, it could be inferred that that PVSOC shall yield better performance than MSOC in an average case where a game may have a maze-like scene to be rendered with occlusion culling, such as in Wolfenstein 3D. For games using larger maze scenes (scenes with a greater number of objects), however, these results suggest strongly that MSOC should be considered if the higher performance is needed by the game developer. For smaller and simpler scenes, however, the data shows clearly that one should expect better performance from PVSOC.

In all three static mazes, PVSOC wrongly labelled more objects as visible than MSOC did. As the number of objects in the maze scenes increased, so too did the number of wrongly labelled objects for both PVSOC and MSOC. This is to be expected. However, the increase in erroneous labels was more pronounced in PVSOC than it was MSOC. The data suggests that in order to minimise the number of erroneous labels for occlusion culling in a maze game, MSOC should be preferred over PVSOC.

MSOC managed to emit zero for every flexibility metric that this research includes. For all of them, a higher value meant lower flexibility. The results indicate very strongly that MSOC is an incredibly flexible algorithm. The flexibility of PVSOC is inferior to MSOC in all three static maze scenes, and this inferiority only gets worse as the maze scenes become more complex. This is because the amount of preprocessing required to allow PVSOC to work has positive correlation with the complexity of the maze scene.

In conclusion, the data does not indicate that either occlusion culling algorithm is superior in all static maze scenes. The algorithm to be preferred is largely dependent on the use-case of the game developer and the nature of the scenes they intend to create. However, regarding maze-scenes, PVSOC should be preferred if one wishes to minimise strain on the CPU. This is because although PVSOC wrongly labels objects as visible more often (which means more objects are rendered per frame, adding strain on the GPU), the CPU spends less time executing the PVSOC implementation than it would MSOC. If, however, the user is rendering complex mazes with many walls, or the performance requirement is not too strict, then they may prefer the use of MSOC. Implementing MSOC into a game-engine means that the game developer can spend more time making their games instead of preprocessing the scene.

Dynamic Maze Scenes

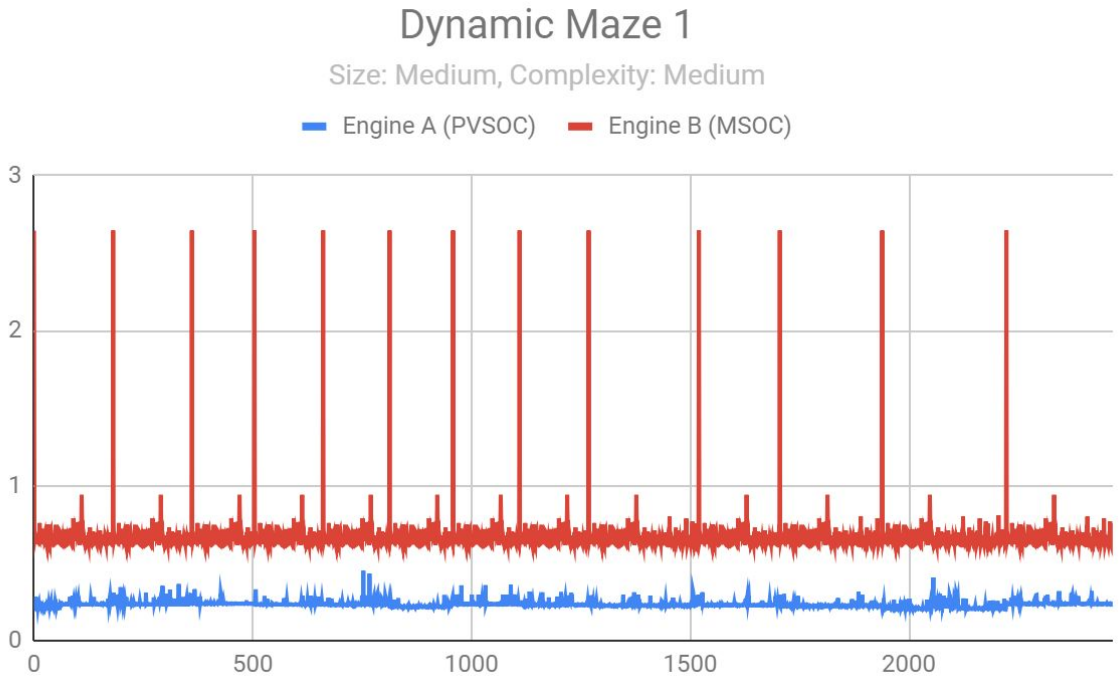
Dynamic Maze 1

Performance

The gathered metrics for Dynamic Maze 1 are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.2342338873	0.6703534657
Standard deviation of execution time per frame (ms)	0.02029638507	0.150260552

All data samples are plotted graphically below.



The performance metrics for Dynamic Maze 1 are very similar to that of Static Maze 1. This is expected, as the scenes are identical except from the varying of the wall heights. Varying the wall heights per frame incurs an overhead, but the overhead is so small that it has no significant effect on the performance metrics. PVSOC is both more consistent and spends less time executing per frame than MSOC. This means that PVSOC outperformed MSOC in this scene.

Accuracy

	PVSOC	MSOC
Number of objects wrongly labelled as visible	219	157
Number of objects wrongly labelled as occluded (Zero means conservative)	17	0

This data highlights a new issue with the PVSOC algorithm; it did not maintain the property of conservative in this dynamic scene. MSOC, however, did remain so. Aside from the varying heights of the walls in this maze, it is otherwise identical to Static Maze 1. This is why the number of objects wrongly labelled as visible are identical. This data suggests that this implementation of PVSOC is not guaranteed to be conservative if the scene is dynamic.

Flexibility

The scene nodes are identical to that of Static Maze 1. The gathered metrics for Dynamic Maze 1 are identical to that of Static Maze 1.

	PVSOC	MSOC
Number of general scene preprocessing steps	13	0

Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

This means that the data suggests that dynamic scenes are equally as flexible as their static counterparts.

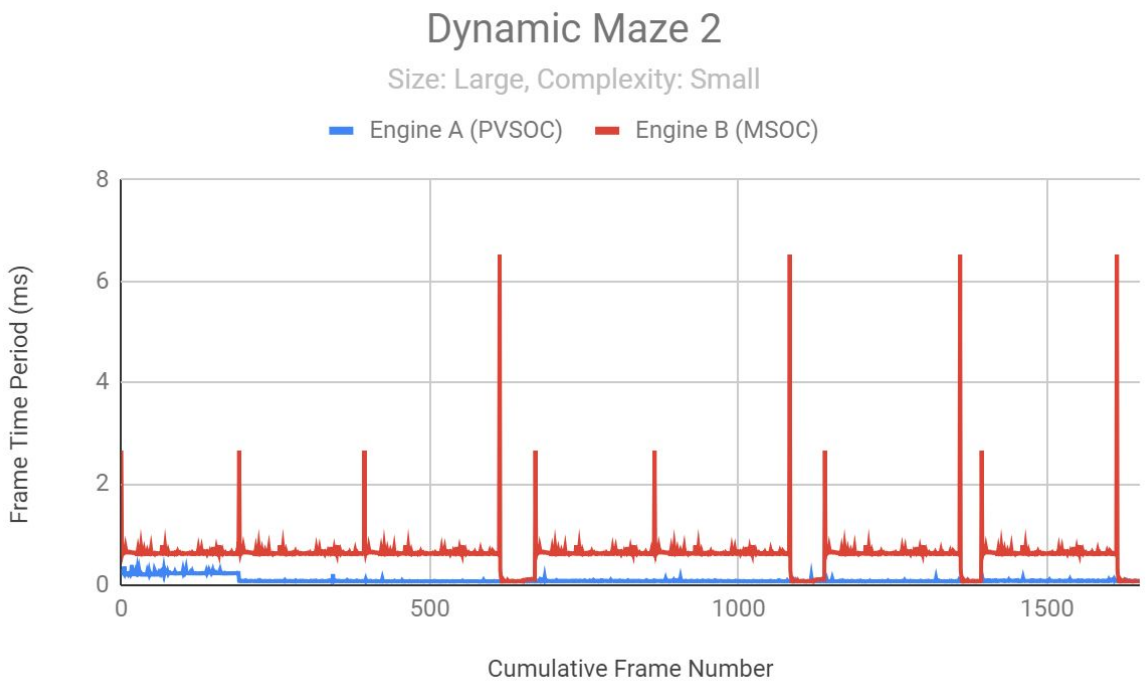
Dynamic Maze 2

Performance

The gathered metrics for Dynamic Maze 2 are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.09958025439	0.5979230769
Standard deviation of execution time per frame (ms)	0.05208312771	0.3656823006

All data samples are plotted graphically below.



Accuracy

	PVSOC	MSOC
Number of objects wrongly labelled as visible	19	16
Number of objects wrongly labelled as occluded (Zero means conservative)	3	0

Similarly to Dynamic Maze 1, the results for Dynamic Maze 2 indicate that this implementation for PVSOC is not conservative for dynamic scenes.

Flexibility

As before, the flexibility metrics for Dynamic Maze 2 are identical to that of Static Maze 2:

	PVSOC	MSOC
Number of general scene preprocessing steps	7	0
Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

This data continues to support the prospect that MSOC is superior to PVSOC regarding flexibility.

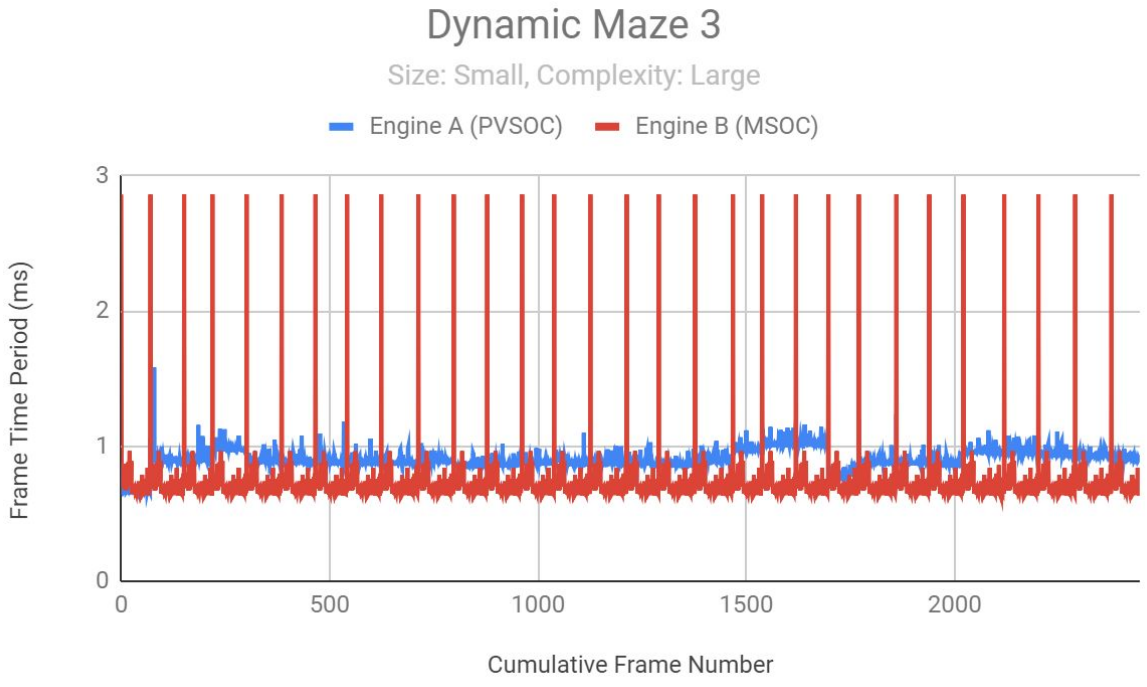
Dynamic Maze 3

Performance

The gathered metrics for Dynamic Maze 3 are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.9032631877	0.7372421268
Standard deviation of execution time per frame (ms)	0.07275307913	0.2454310169

All data samples are plotted graphically below.



Accuracy

	PVSOC	MSOC
Number of objects wrongly labelled as visible	219	157
Number of objects wrongly labelled as occluded (Zero means conservative)	69	0

In this scene, PVSOC wrongly labelled many objects as occluded. Due to this, objects would randomly disappear from view when moving across nodes. This occurrence, caused by a lack of conservative occlusion culling implementation, would completely disrupt the immersion of a player if it were to be used in a maze game, for example.

Flexibility

The metrics are identical to that of Static Maze 3:

	PVSOC	MSOC
Number of general scene preprocessing steps	30	0
Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

Dynamic Maze Conclusion

The metrics produced by the dynamic maze analysis complements many of the conclusions drawn by the static maze analysis. Namely, MSOC is consistently outperformed by PVSOC due to a large initial overhead, but as the number of objects in the scene increases, the overhead becomes miniscule compared to the growth rate. This leads to MSOC consistently outperforming PVSOC if the scenes contain a large number of objects.

The data gathered from the dynamic mazes highlights a huge flaw with this implementation of PVSOC; it is not conservative in dynamic scenes. This is to be expected with this particular implementation of PVSOC, because the node boundaries are calculated **once**, when the scene is imported and all the walls are at their initial heights. As the heights of the walls vary, the potentially-visible sets should change, but they do not as the node boundaries are not recalculated.

In response to this, I amended the algorithm implementation to recalculate the scene boundaries every frame. This did cause PVSOC to become conservative in dynamic scenes. However, the sheer amount of frequent recalculations ended up incurring a massive performance loss. On the specified platform & hardware, Maze 3 could not yield above 5 frames per second. As the performance was clearly unacceptable, this amendment was ultimately discarded and is therefore not otherwise included in this research. Indeed, the performance was so poor, that the process of gathering metrics became too difficult to do properly. Thus, no results for this amendment can be provided.

Other Scenes

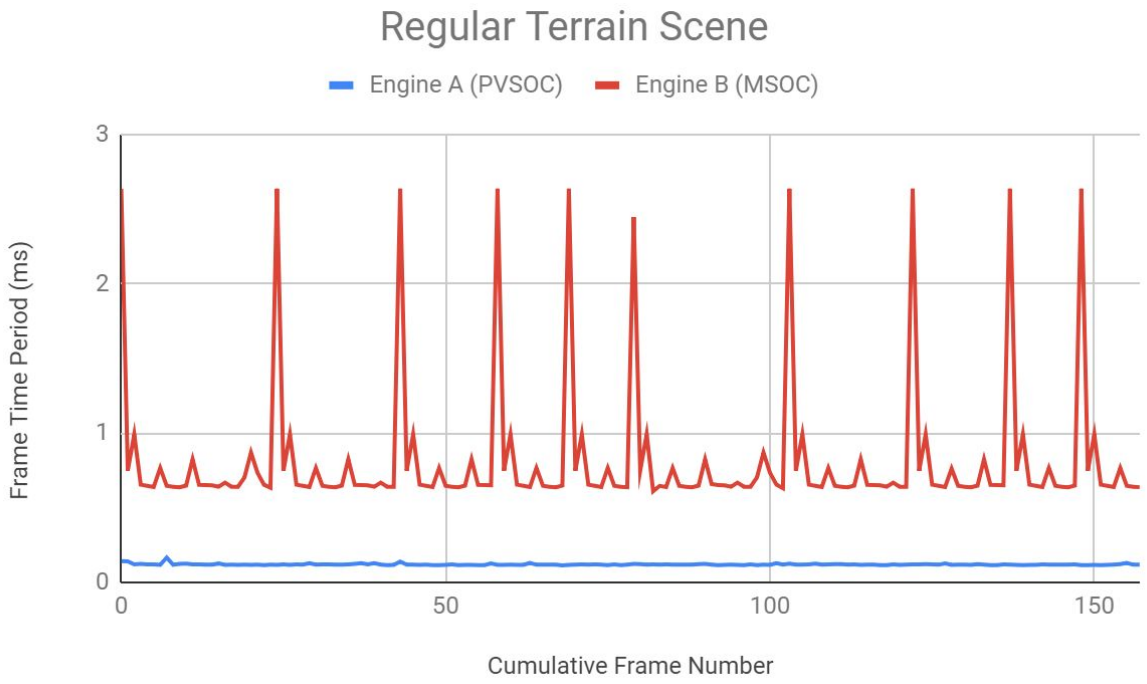
Regular Terrain

Performance

The gathered metrics for the Regular Terrain Scene are as follows:

Correct to 3 significant figures	PVSOC	MSOC
Mean execution time per frame (ms)	0.1197770701	0.8058216561
Standard deviation of execution time per frame (ms)	0.005371576102	0.4573399749

All data samples are plotted graphically below.



This data shows that PVSOC outperformed MSOC in this scene more than in any of the static or dynamic maze scenes. This refutes the earlier claim that the performance of MSOC improves relatively to PVSOC as the number of objects in the scene increases. The Regular Terrain scene has a larger number of objects (144) then any of the other scenes (the second largest was Maze 3, with a total of 104 objects).

This means that there are other variables which affect the performance of MSOC relative to PVSOC other than the number of objects. It is necessary to remember that these performance metrics do not take into account the rendering time, so the accuracy of the occlusion culling is not affected at all nor is affected by these performance metrics. The reason for these results is subtle yet obvious when noticed: In the maze scenes, each object was an indexed cube mesh, consisting of eight faces. Each object in this scene is a copy of a 'wave' mesh which is available in the resources directory of Engine A and Engine B (res/runtime/models/wave.obj). This mesh is far more complex than the cube mesh.

For Engine A, this affects the performance of the preprocessing section of PVSOC; it is during this time when the node boundaries are calculated, which is affected by the complexity of the mesh. For Engine B, however, each object is software-rendered into the depth-buffer used by MSOC every frame. This means that the higher number of primitives in these objects massively increased the software-rendering time for MSOC, causing the average frame time of MSOC to increase.

The fact that the meshes were identical in all other scenes, yet far more complex in this scene, meant that the results for this scene is skewed. If the terrain was made instead with the cube meshes, the performance metrics would be alot more realistic. This was a limitation in my scene design process; fair results demand that all other conditions aside from the independent variable remain constant. In this case, the independent variable was the composition of the scene, but the condition of the mesh varied.

Accuracy

For PVSOC, this scene happened to be designed almost perfectly, due to its regularity and geometric simplicity.

	PVSOC	MSOC
Number of objects wrongly labelled as visible	0	360
Number of objects wrongly labelled as occluded (Zero means conservative)	0	0

This scene activated a special case for PVSOC; each object has its own node consisting of itself, and no potentially-visible-set. This was because the camera was only being sampled due to it being in a valley. While in a valley, no other objects would be visible. Because of this, only 1 of the 144 objects would be visible at a time.

MSOC however, would simply render all objects as occluders into its depth-buffer and query them as normal. Due to a constant leakage in the depth-buffer, a known drawback of the MSOC algorithm, it would sometimes wrongly label an object as visible. However, MSOC was designed so that this leakage would not stop it from being conservative. It is for this reason why MSOC falls behind regarding accuracy results.

Flexibility

The metrics for the regular terrain scene are as follows:

	PVSOC	MSOC
Number of general scene preprocessing steps	144	0
Number of object-specific preprocessing steps	1	0
Number of additional metadata per object	1	0

As stated previously, this scene was unique in that it allowed PVSOC to have one node per object. While this resulted in excellent metrics for accuracy, it has strongly highlighted the flexibility drawback.

Overall Conclusion

Performance

The metrics gathered from the varying types of scenes indicate that there exist additional unknown variables which affect the performance of PVSOC compared to MSOC. The execution time of MSOC appears to be closely related to the number of primitives in the scene (meaning the total number of triangles in all objects in the scene). This is also an intuitive observation; rasterizing additional primitives into a hierarchical depth buffer will require more processing time.

However, the data does not indicate that the performance of PVSOC is so simply related to the number of primitives in the scene; it is likely that the relationship is more complex. For example, the composition of objects a scene affects the node boundaries and the potentially-visible-sets in the scene. In the maze scenes, mazes with more objects resulted in more possible nodes which the camera may reside in, reducing performance. In the terrain scene, however, the number of objects was also large, yet PVSOC vastly outperformed MSOC. However, the data from the regular terrain scene is less useful, as it used different meshes and thus is not a truly fair comparison with any of the maze scenes.

In summary, there is no clear identifiable circumstance which would guarantee if MSOC would be a more performant technique than PVSOC. Although the number of primitives has a clear relationship on the performance of MSOC, there are other variables which affect PVSOC which may vary strongly between different types of scenes.

Accuracy

In scenes without moving objects, both implementations of PVSOC and MSOC were conservative, meaning that they never wrongly labelled an object as occluded. In scenes where there existed at least one moving object, however, PVSOC would not be conservative because the node boundaries and potentially-visible-sets would become incorrect.

Therefore, in dynamic scenes, the data indicates that MSOC should always be preferred, because a conservative occlusion culling algorithm is a necessity. However, additional measures can be taken to ensure that PVSOC remains conservative even in dynamic scenes. This is visible in the Unity engine, where its occlusion culling remains conservative & highly accurate even as objects in the scene move.

In static scenes, there is no clear superior choice. If a game is utilising maze-scenes of a similar format to the ones used in the analysis of this project, then the choice becomes clearer: PVSOC should be preferred if the number of walls in the maze is at least smaller than that of Maze 1 (around 41 or below). With mazes with more walls such as in Maze 3 (104 or more), MSOC becomes the preferred choice as the overhead of the MSOC execution becomes minimal compared to the growth-rate of both algorithms, especially PVSOC.

Flexibility

There is a clear trend in the flexibility metrics for PVSOC and MSOC. For the 'Number of general scene preprocessing steps', the metric for PVSOC was equal to the number of nodes partitioning the scene. For MSOC, it was always zero as there were no preprocessing steps at all.

For the 'Number of object-specific pre-processing steps', the metric for PVSOC was always 1. This is because every object would be allocated to exactly one node. For MSOC, this was of course always zero because there were no preprocessing steps at all.

For the 'Number of additional metadata per object', every object for PVSOC needed to store the name of the node in which it belonged to. As each object was allocated to one node only, this resulted in the metric always being one. For MSOC, there was no pre-runtime component at all; the entire algorithm logic and functionality is executed at runtime. This meant that there was no need for any additional metadata per object, as rendering the object into the depth-buffer required nothing more than the default data elements it came with; position, rotation, scale, and its mesh.

Thus, regarding flexibility, all data in this analysis suggests that MSOC is a superior algorithm to PVSOC. To provide evidence for this for the general case is hard, as the number of scenes used for analysis was not that high, nor were the number of different types of scenes. Although the pattern for MSOC is clear that the algorithm has no preprocessing requirements for scenes at all. Knowing also the logic of MSOC being completely executed at run-time, logic suggests that MSOC should support any and all scenes without any amendments, so long as the objects can be properly sent to MSOC's rasterizer. This may not be the case under some conditions, such as if the underlying primitives used for the mesh are not supported by it. However, under typical circumstances for triangulated meshes, there is no evidence here that MSOC will require amendments to accommodate any scene.

Summary and Reflections

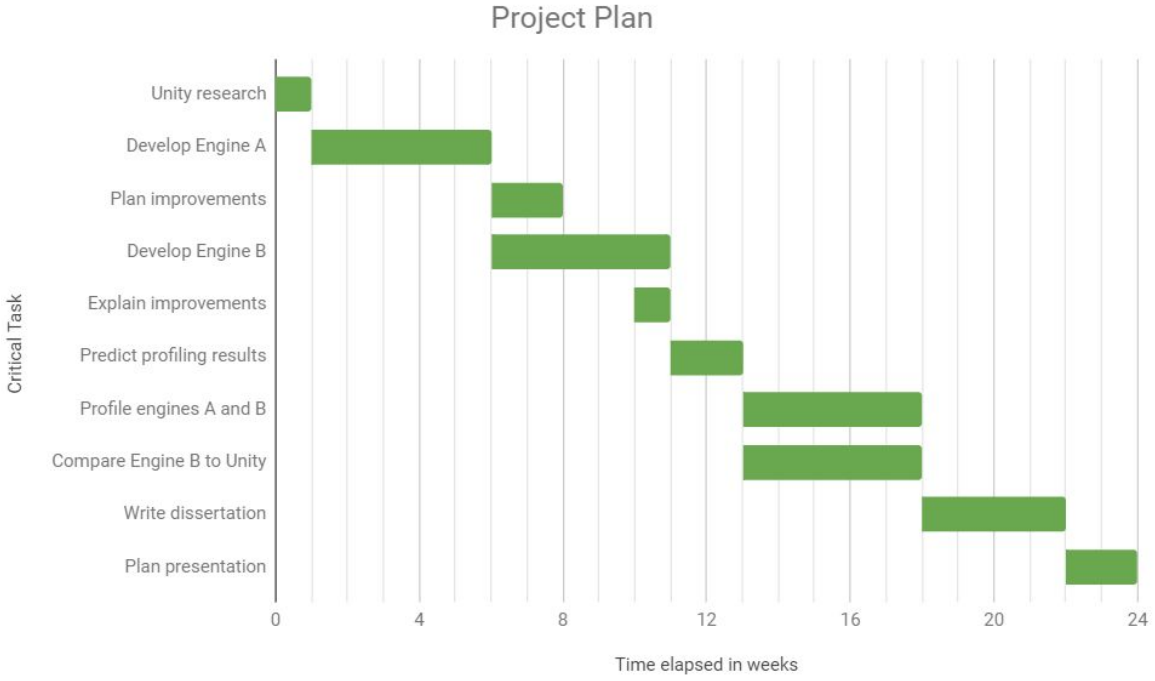
For performance, there is not enough data in this research to indicate under which circumstances MSOC outperforms PVSOC, nor the other way round. If tessellation and objects with large numbers of primitives exist in the scene (such as a dragon model in a video game), the performance of MSOC is likely to suffer such that PVSOC becomes the more performant technique.

Regarding accuracy, MSOC is an excellent choice in scenes with moving objects, as it does not suffer from the loss of the conservative property as PVSOC does. With static scenes, the accuracies yielded by both techniques cannot be so easily predicted. However, there exists a possible solution which boasts the benefits of both MSOC and PVSOC. It is possible that PVSOC can be performed on a given scene. For all objects in the potentially-visible-set of the node containing the camera, MSOC can be performed to issue occlusion-queries for that specific subset of the scene. This means that fewer objects need be rasterised into the MSOC depth buffer, while retaining the conservative property of MSOC in even dynamic scenes. An excellent extension to this research would be to identify such a technique, as it may yield excellent performance and accuracy metrics despite the loss of flexibility incurred through the use of PVSOC.

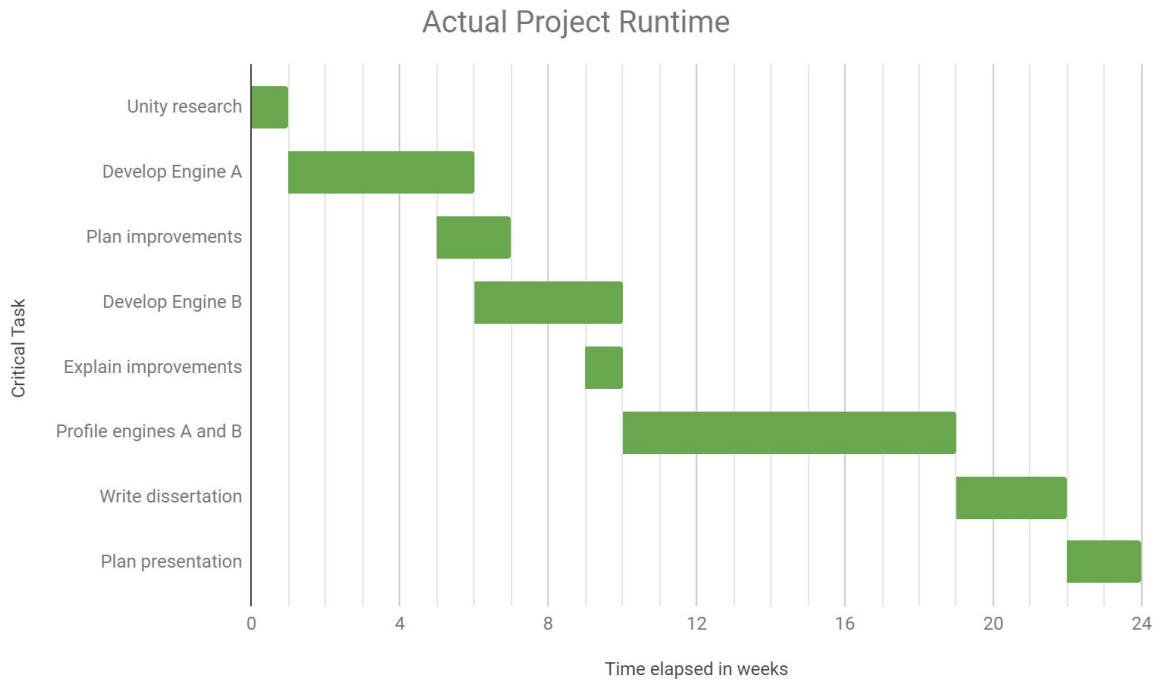
For game engines favouring maximum flexibility however, this research asserts that MSOC is the superior choice in all circumstances. This is however, subject to exceptions. For example, Unity may not benefit as much from using MSOC, because the preprocessing of PVSOC (which was responsible for the lack of flexibility) is automated by the engine, thus retaining much flexibility.

Project Management

Below is a gantt chart depicting the original plan for the project.



In regards to time management, most of the tasks were accomplished on-time and as planned. However, there were a few instances where this was not the case. Below is the gantt-chart representing the tasks as they actually took place.



This chart is different in several respects:

- The task 'Predict Profiling Results' has been removed because it was deemed irrelevant to the value of the research.
- The task 'Compare Engine B to Unity' has been removed because it was since decided that Engine B should be compared to Engine A only, and comparing it to Unity aswell would not have added value to the research.
- The task 'Develop Engine B' had a duration one week shorter than planned. This was because the Masked Software Occlusion Culling algorithm was used via an external library and thus required no implementation from myself.
- The task 'Profile engines A and B' had a duration of nine weeks as opposed to five weeks as planned. This was due to the fact that when making the plan, the quantity of metrics which I expected to gather were far less than that which was actually gathered.
- The task 'Write Dissertation' had a duration of three weeks as opposed to four weeks as planned. This was because I had already performed much of the critical analysis during the 'Plan improvements' and 'Explain improvements' tasks, whose value I had underestimated in the original plan.

Contributions and Reflections

This research considers MSOC a rather favourable algorithm which should be seriously considered by game-engine developers when creating their technologies. The data indicates that there exist many circumstances where it may be a better alternative for achieving occlusion culling rather than an approach based upon potentially-visible-sets of objects such as PVSOC. As MSOC is a newer algorithm and thus has not received a vast amount of attention from the game-engine industry, support for it is increasing. For example, the Unreal Engine has an option for enabling software-occlusion-queries, utilising an approach very similar to that of MSOC (Epic Games, Software Occlusion Queries for Mobile). In addition, a software-based approach has been used in the Frostbite engine for several years (Collin D., 2011). It would not be unreasonable to expect support for MSOC or a SOC-based approach to grow even more over time.

The plan for this project was a useful guideline to follow throughout the research process. However, it became less useful as changes were made during the later stages of the project. For example, the removal of predictions would have resulted in a duration of two weeks of inactivity, if the plan were not amended. Beginning later tasks a few weeks earlier

prevented this time from being wasted, but ultimately resulted in the plan becoming less and less useful as more things were changed. This was the main risk incurred by following the plan too strictly. Despite this, it remained a useful aid for time-management.

The research conducted throughout the project is highly extensible; performing the same analysis with other occlusion culling techniques, such as GLQOs described earlier, would allow more algorithms to show their potential under certain circumstances that this project provides, such as in dynamic-maze scenes. The methodologies however, specifically in regard to gathering performance metrics would require amendments to fairly quantify the performance of a GPU-based approach such as GLQOs.

Bibliography

Collin D. (2011). *Culling the Battlefield: Data Oriented Design in Practice*. Slide 41/55. DICE. Available at: https://www.ea.com/frostbite/news/culling-the-battlefield-data-oriented-design-in-practice

Epic Games. *Performance and Profiling Overview*. Available at: https://docs.unrealengine.com/en-us/Engine/Performance

Epic Games. *Software Occlusion Queries for Mobile*. Available at: https://docs.unrealengine.com/Engine/Rendering/VisibilityCulling/SoftwareOcclusionQueries

Gregory J. (2018). *Game Engine Architecture, Third Edition*. p13. Available at: https://books.google.co.uk/books?isbn=1351974270

Hasselgren J., Andersson M., Akenine-Möller T. (2016). *Masked Software Occlusion Culling*. Intel Corporation. p1-6. Available at: https://software.intel.com/sites/default/files/managed/ef/61/masked-software-occlusion-culling.pdf

Lengyel E. (2010). *Game Engine Gems, Volume One*. p10. Available at: https://books.google.co.uk/books?isbn=1449657931

Lustig D., Martonosi M. (2013). *Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization*. p1-4. Princeton University. Available at: http://mrmgroup.cs.princeton.edu/papers/dlustigHPCA13.pdf

Nirenstein S., Blake E., Gain J. (2002). *Abstract Exact From-Region Visibility Culling*. p2. Eurographics Association. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.7204

Reinders J. (2013). *Intel® AVX-512 Instructions*. Available at: https://software.intel.com/en-us/blogs/2013/avx-512-instructions

Appendices

- Relevant GitHub link containing references to repositories for Topaz, Engine A and Engine B: https://github.com/Harrand/Dissertation
- Accuracy Metrics extracted from screenshots of the framebuffer of given frames in both Engines A and B. Although the screenshots are too large to be included in these appendices, a text file is present which contains a URL to such images if they are required.
- Performance Metrics extracted from the TimeProfiler class in Topaz and thus available in Engine A and Engine B.