Real Time Graphics Engine Improvement Through Applied Culling Techniques

William Andrew
Games Academy
Falmouth University
Falmouth, United Kingdom
WA278394@falmouth.ac.uk

***Abstract*—This is Sokol's example, use it as a template**

(open) In this paper, we explore the effectiveness of measurement tools and their usefulness determining issues with artificial intelligence pipeline integration. Given the already complex nature of artificial intelligence pipelines, the ability to assure the effective configuration of Mixed-initiative artificial intelligence(MIAI) tools, is imperative to the determination of the distribution of resources when developing these systems.

(Challenge) To determine the effectiveness of our MIAI tool we looked at traditional survey tools to help identify areas of improvement for our own pipeline, the Creativity Support Index (CSI) is regarded as a useful tool for quantitively assessing the usefulness of creative software applications. However, real world validation of the Creativity Support Index as a measurement tool for MIAI tools has not yet been established.

(action) We developed a MIAI tool, Melete, in which we tested the individual components of this MIAI tool using the CSI to determine the effectiveness of the individual components of an MIAI tool. Two cohorts of undergraduate students tested Melete. The students participated in two separate studies exploring the use of Melete in a creative task for level design. These studies encompassed six individual conditions; each condition evaluated the individual components of Melete, using the CSI, with a combined sample size of 324 responses to the CSI.

(resolution) Our analysis of the CSI scores indicates that while the CSI is a valuable tool, there are several areas for improvement. We make specific recommendations to improve the CSI including, the disentanglement of measures and the development of a robust set of questionnaires based on a factor analysis. This work represents our contribution towards developing a reliable and practical metric for developing MIAI tools. Our work has analysed a widely recognized and utilized measure, the CSI and provided recommendations for future work within the MIAI and AI pipeline measurement research space.

I. Introduction

What is is, and why its important Who is going to find it helpful?

II. Literature Review

A. Graphics pipeline/rendering process

The graphics pipeline is the process responsible for transforming 3D coordinates into 2D pixels in screen space. It has five different steps [1], each of which are briefly outlined and explained:

- Vertex Shader - this takes a single vertex as an input. This is repeated until primitive assembly takes all the vertices and assembles them into primitive shapes
- Rasterization - this stage maps the primitive shapes to the corresponding on-screen pixels
- Fragment shader - This calculates the colour of the pixels. It often also contains data such as lighting and shadows, which is about the 3D environment, and not the primitive.
- Output merging - The depth value of the fragments are checked.
- Display - Displays the image in screen space.

In most graphics applications, the only programmable parts, and therefore the only parts that should mainly be worried about are the Vertex and Fragment shaders [2].

B. Graphics APIs

An API a piece of software which acts as a contract between applications, [3] allowing them to talk to each other. While many different types of APIs exist, such as Web and RPC APIs, the experiment which this paper covers will use a Graphics API, which allows rendering graphics on the GPU to be more feasible. [4] There are many well known graphics API's, notably Vulkan, Direct3D and OpenGL, the last of which will be used for this artefact. OpenGL is a 3D Graphics API which allows users to write code, applications and shaders which produce graphics in both 2D and 3D. [5] While newer graphics API's such as Vulkan are far more power efficient on the CPU, it has a far steeper learning curve than OpenGL [6]. On the other hand, OpenGL is a relatively higher level API, and has been the most widely adopted API [7] since its release in 1991. There is therefore more support available for OpenGL, which will help in supplementing limited API knowledge.

C. Mesh Generation

According to Marshall Bern & Paul Plassmann, a mesh is a discretization of a geometric domain into small simple shapes [8]. Triangles in particular are used intensively to create these meshes [9] because of their simplicity, allowing for flexibility and efficiency [10]. The polygons that make up these meshes are each comprised of vertices, which in terms of regular geometry, is simply a point or "corner" of a 3D shape. In computing, a vertex is a data structure which allows for

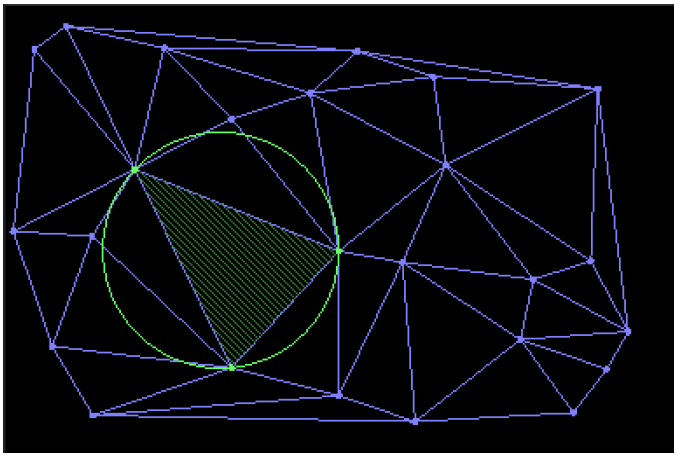

Fig. 1. An example of Delaunay triangulation used on a complex mesh. Note that the three vertices of the highlighted triangle are always on the circumference of the circle, and never inside it.

manipulating the polygons which make up a mesh [11]. The data contained in them is their position, which they always have, and is typically stored as a three dimensional vector, like so:

```
vec3 vertices[] =
{{x1, y1, z1},
{x2, y2, z2},
{x3, y3, z3}};
```

This example code snippet shows the logic for the vertices of a triangle, where x, y and z are each floats, representing the x, y and z coordinates of the vertexes location in 3D space. The process of converting a complex mesh into simple triangles is known as triangulation, and can be achieved with a variety of algorithms, the most common of which is Delaunay Triangulation [12]. Which applies the method of splitting the plane of the mesh into a series of triangles, which have none of their vertices lying within it's circumcircle [13], as shown in figure 1. [14]

D. Frustum View Culling

The first of the culling techniques used for this experiment is frustum view culling. A frustum is a shape consisting of six planes, in which only the closest and furthest planes (the near and far planes respectively) are parallel to one another [15]. The result is a pyramid-shaped volume projected outwards from the camera as shown in figure 2 [16]. The program will then only render the triangles of meshes inside the view frustum [17], leading to the GPU or CPU rendering significantly less objects in an environment, increasing performance. This happens because objects outside of the view box do not undergo the entire graphics pipeline process, skipping computationally expensive parts such as rasterization and lighting [18]. Furthermore, basic frustum view culling has been further developed and optimized by combining it with octree data structures, improving culling efficiency greatly [19]. However, the complexity of modern day environments is only increasing,

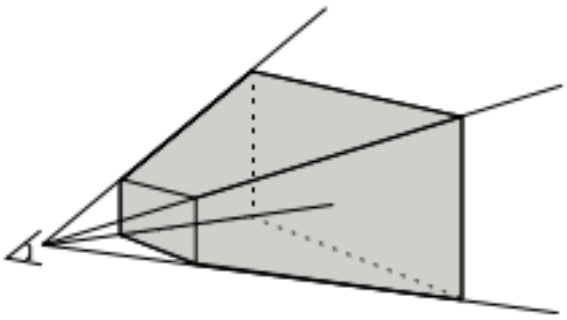

Fig. 2. The view frustum projected out from the camera view.

resulting in un-optimized culling methods being unable to keep up [20], a clear drawback of this culling technique . This experiment will aim to negate this drawback somewhat by implementing two other culling techniques alongside frustum view culling, which should allow for the program to run more efficiently, without the use of an optimized frustum view culling technique.

E. Back Face Culling

The second culling technique used for this experiment is back face culling. In a 3D environment, when the user looks towards an object, there will always be some faces facing away from the player, meaning they cannot be seen. Despite this, they are still rendered, which wastes memory resources [21]. Back-Face culling finds the parts of the model that cannot be seen from the users viewpoint, and does not render them. This is achieved by calculating the dot product between the normal and the view point vector. [22]. If the result is greater than or equal to zero, it is therefore pointing away from the viewpoint, and cannot be seen by the user. Conversely, if it is less than or equal to zero, it's normal will be pointing towards the viewpoint and should be rendered. The following pseudocode can show this, using a triangle as an example:

Algorithm 1 A back-face culling algorithm

$v1Dir \leftarrow v1 - v0$
$v2Dir \leftarrow v2 - v0$
$norm \leftarrow crossProduct(v1Dir, v2Dir)$
$norm.normalize()$
$viewPoint \leftarrow (x, y, z)$
for $each\ face\ in\ the\ mesh$ **do**
  if $dotProduct(view, norm) \geq 0$ **then**
    $renderBackFaces \leftarrow false$

Back face culling is an excellent technique to use for graphics optimisation, because on average, half of the polygons on a mesh will be back facing [23] only half of the polygons on the mesh will be rendered at one time. If this algorithm is applied to all meshes at once, then theoretically, half of all

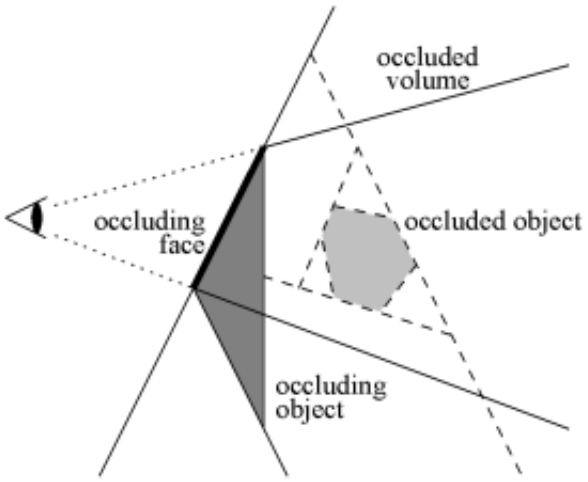

Fig. 3. The view frustum projected out from the camera view.

the polygons in the environment will be culled at once, significantly reducing computational load. When combining this with frustum view culling, the total number of triangles can be reduced to upwards to 80% [24] - a drastic improvement.

F. Occlusion Culling

The final culling technique used for this experiment will be occlusion culling. This culling technique works by culling objects that cannot be seen. For objects outside of the cameras field of view, this functions the same as frustum view culling. Where occlusion culling differs, is that it also culls objects that are obscured by others [25], as show in in figure 3 [26]. An common approach to implementing occlusion culling is by utilizng the Z-buffer (also known as the depth buffer), which stores the value of the objects depth [27]. By rendering the environment front-to-back, the depth values of objects in the environment will be added to the z-buffer in order of closest to furthest. This allows us to create a sort of list for occlusion culling [28], which can then be combined with the camera position to tell which objects should and shouldn't be rendered, which can be expressed in pseudocode:

Algorithm 2 An occlusion culling algorithm

$cameraPos \leftarrow (x, y, z)$
$depthVals \leftarrow list < float >$
for $each\ value\ in\ depthVals$ **do**
  if $cameraPos > depthVals[i-1]\ AND\ cameraPos.z <= depthVals[i]$ **then**
    $renderCurrentObj \leftarrow true$

III. Research Questions

A. Findings

1) Graphics API: The preliminary research into graphics API's suggests that the API of choice for this experiment, OpenGL, will run at a lower frame rate than Vulkan and Direct3D [29]. However, OpenGL is easier to learn, and has been the more widely adopted graphics API for longer. This should mean that it is more stable, and has more widely available support.

2) Mesh Generation: Triangles are considered the most efficient and effective primitive to use when it comes to mesh generation. They can be combined to create many other primitive shapes such as squares and even circles. Triangles can also be generated from more complex meshes through the use of Delaunay Triangulation [30], which will likely be done for the meshes used in this experiment.

3) Frustum View Culling: The research on Frustum View Culling shows that it is an excellent culling technique with lots of flexibility and variation for the user depending on the performance of their machine. The Far plane of the frustum can be dynamically adjusted to increase or decrease the viewing distance, while the side views can be adjusted along side the view cameras field of view to give a more narrow or wide viewpoint. This therefore allows more or less objects to be culled depending on whether the user wants to prioritize performance or quality.

4) Back Face Culling: The research into back-face culling revealed it to be a very promising technique. As previously outlined, when implemented alone it can cull approximately half of the triangles on average. When implemented alongside the other culling alogorithms, it can drastically decrease the number of triangles, therefore drastically increasing the performance of the program.

5) Occlusion Culling: Occlusion culling appears to be the most complex of the three algorithms chosen for this experiment, making extensive use of the Z-Buffer, and relying more heavily on variables such as the camera viewpoint. It also appears to benefit from environments with larger meshes which obscure others from view.

B. Hypotheses

The research question derived from the following research will be: "To What Extent Can the Performance of a Real Time Graphics Engine be Improved Through Combined Culling Techniques?" With the following hypotheses being derived:

1) "Applying culling algorithms to an environment with a high poly count will increase CPU GPU performance, frame rate and frame consistency, while decreasing frame latency." This should happen because the CPU/GPU is rendering less vertices per frame, which will display less on screen.
2) "The total number of triangles being rendered at once in the environment will decrease by a factor of approximately 80% - 90%." A statistic of 70% - 80% was achieved by C.Lee et al by applying two culling algorithms [24], therefore it is reasonably to assume that a greater mean number of triangles can be culled with three culling algorithms.
3) "When applied in isolation, back-face culling will improve performance the most and cull the most trian-

gles. Occlusion culling when applied in isolation will improve performance the least." This is working off the assumption that back-face culling should be culling approximately half of all triangles in the environment, whereas frustum view culling and occlusion culling depend on the viewing direction of the user.

IV. Computing Artefact

A. Description

The artefact created for this experiment will be a series of three dimensional OpenGL environments, which will contain the following:

- Varying high poly meshes scattered around the cameras initial viewpoint. These are the meshes which will be affected by the various culling algorithms used. Statistics such as framerate, GPU CPU performance, and triangle count will be affected by these meshes, and will differ slightly depending on the variety of meshes in each environment.
- An interactable camera. This will be the users primary method of interaction and navigation in the environment. It will contain the code for keyboard and mouse movement, zoom field of view interaction, and if necessary, any postprocessing effects. The camera will also contain the code for each of the culling algorithms.

The environments themselves will have simple interactivity, similar to that of a standard graphics simulation. The user will be able to navigate around the environment using key inputs, and look around using the mouse - a necessity for the culling algorithms to work in real time as intended.
This experiment will be carried out only on computers which use NVIDIA GPU's, which is to avoid any potential performance differences despite the same algorithms being used. This avoids inaccuracies created by hardware requirements, allowing for the sole focus of the experiment to be on the algorithms used, and the environments they're used in. The data collected will be a regular intervals during runtime. This allows for any unexpected performance drops or spikes to be recorded accurately. Fig 4 in the appendix describes the process by which the artefact will work, with Fig 5 (TODO: Add Fig 5) showing the class structure of the project. The following project's repository can also be found here:
https://github.falmouth.ac.uk/GA-Undergrad-Student-Work-24-25/COMP302-WA278394-2204080.git

B. Development Methodology

This artefact will make great use of version control, with four separate branches being made use of - One for each culling algorithm to be applied individually, and then a fourth for having all three applied at once. This will ensure that when individual culling algorithms are applied, there are no variables for other algorithms interfering. This experiment process will also make great use of Agile software development methodology, which advocates for a reflective and adaptive development process. [31] This is ideal for an experimental piece of software, because development of the artefact will remain flexible should anything unexpectedly change. Agile also advocates for an iterative approach to software development, an ideal approach because new features to enhance the experiment can be added (or removed if necessary) to fit the scope of the project.

C. Quality Assurance

1) Construct Validity: To ensure high quality research, this experiment will measure the mean values for various types of data, which will measure the efficiency and performance of the various applied culling algorithms. The primary statistic chosen will be the mean number of triangles/polygons being sent to the GPU to render, with the only user input in the environment being a single full rotation. This ensures that the entire environment is being viewed and culled, and specific meshes with higher or lower poly counts aren't excluded or included from the culling process any more than others. Other secondary factors which will be used to measure efficiency include mean memory usage, mean CPU and GPU usage, and mean frame rate. CPU, GPU and memory usage statistics can be gathered through the use of Microsoft Visual Studio's profiling tool [32], while frame rate and polygon count will be gathered through output code in the project, which will also be exported to a separate file and analysed using R code. [33]

2) External Validity: This experiment has excellent external validity because of the widespread common usage and need for graphics simulations in various industries, such as Games development, CAD modeling software and even computer-aided content creation and medical screening [34]. Games and Modeling software will often be required to render thousands of high poly meshes at one time, with thousands more being active in the environment at one time. This often results in the requirements to run the software being very demanding, such as Unreal Engine 5 made by Epic Games, who define a "typical" system used to run Unreal Engine 5 containing an RTX 3080 GPU, 128 Gigabyte RAM, a 4 Terabyte SSD and an "AMD Ryzen Threadripper Pro" processor, which according to them, constitutes a "reasonable guide to developing games" in their engine [35]. Applying culling techniques in real time will be helpful for software such as this, because it will allow users with lower spec machines than the recommended to run the game engine with an increased framerate, and lower memory and GPU usage. This would be especially useful for simulations of a medical or military nature, in which the speed of the program is paramount, because it could have potentially life threatening consequences if un-optimized.

3) reliability: The data collected from this experiment can be easily repeated, and should yield similar accurate results every time. The three culling algorithms applied to the simulations will maintain the same parameters throughout each environment, and therefore should cull a similar amount of triangles each time, with the main factor being the direction in which the user looks in. Furthermore, the environments will

each be pre-rendered, which allows for greater specificity and variety for testing the culling algorithms in.

4) replicability:

V. Methodology

A. Research Philosophy

Clarify my philosophical position. Find well known ones from the lecture materials and see which one looks best.

B. Project Structure

How am i going to use my artefact to carry out the experiment. What process will i go through? How i will collect, analyse and use data to attain my conclusions. Contain a detailed explanation of what i did, how i did it and why i did it Include G* Power analysis and R Code for statistical analysis. (How i will use it)

C. Variables Libraries

A description of what the dependent, independent and control variables are. Justify why. A description of the libraries used for the experiment. Again, description of what they do/what they are and why im using them.

References

[1] G. Wetzstein, "The graphics pipeline and opengl." Available: https://stanford.edu/class/ee267/lectures/lecture2.pdf. unit cote: EE267, Stanford University,.

[2] J. de Vries, "Learnopengl, hello triangle." Available: https://learnopengl.com/Getting-started/Hello-Triangle. accessed 03/12/2024. [Online].

[3] F. N. Iqbal, "A brief introduction to application programming interface (api)," 2023.

[4] A. Mikkonen, "Graphics programming then and now: How the ways of showing pixels on screen have changed," 2021.

[5] D. Mistry, "Graphics processing unit with graphics api," 2011.

[6] M. Lujan, M. Baum, D. Chen, and Z. Zong, "Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server," in *2019 International Conference on Computing, Networking and Communications (ICNC)*, pp. 777–781, IEEE, 2019.

[7] L. Stemkoski and M. Pascale, *Developing graphics frameworks with Python and OpenGL*. Taylor & Francis, 2021.

[8] M. W. Bern and P. E. Plassmann, "Mesh generation.," *Handbook of computational geometry*, vol. 38, 2000.

[9] R. W. Sumner and J. Popović, "Deformation transfer for triangle meshes," *ACM Transactions on graphics (TOG)*, vol. 23, no. 3, pp. 399–405, 2004.

[10] M. Botsch, M. Pauly, C. Rossl, S. Bischoff, and L. Kobbelt, "Geometric modeling based on triangle meshes," in *ACM SIGGRAPH 2006 Courses*, pp. 1–es, 2006.

[11] C. Esperana and H. Samet, "Vertex representations and their applications in computer graphics," 1998.

[12] L. Chen and J.-c. Xu, "Optimal delaunay triangulations," *Journal of Computational Mathematics*, pp. 299–308, 2004.

[13] G. Eder, M. Held, and P. Palfrader, "Parallelized ear clipping for the triangulation and constrained delaunay triangulation of polygons," *Computational Geometry*, vol. 73, pp. 15–23, 2018.

[14] I. Henry, "Visualizing delaunay triangulation." Available: https://ianthehenry.com/posts/delaunay/. accessed 26/11/2024. [Online].

[15] U. Assarsson and T. Moller, "Optimized view frustum culling algorithms for bounding boxes," *Journal of graphics tools*, vol. 5, no. 1, pp. 9–22, 2000.

[16] U. Assarsson, "View frustum culling and animated ray tracing: Improvements and methodological considerations," *Department of Computer Engineering, Chalmers University of Technology, Report L*, vol. 396, 2001.

[17] M. S. Sunar, A. M. Zin, and T. M. Sembok, "Improved view frustum culling technique for real-time virtual heritage application.," *Int. J. Virtual Real.*, vol. 7, no. 3, pp. 43–48, 2008.

[18] C. Wang, H. Xu, H. Zhang, and D. Han, "A fast 2d frustum culling approach," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 3, pp. V3–414, IEEE, 2010.

[19] J.-W. Zhou, W.-G. Wan, B. Cui, and J.-C. Lin, "A method of view-frustum culling with obb based on octree," in *2007 IET Conference on Wireless, Mobile and Sensor Networks (CCWMSN07)*, pp. 680–682, 2007.

[20] M. Su, R. Guo, H. Wang, S. Wang, and P. Niu, "View frustum culling algorithm based on optimized scene management structure," in *2017 IEEE International Conference on Information and Automation (ICIA)*, pp. 838–842, 2017.

[21] M. S. Sunar, T. M. T. Sembok, and A. M. Zin, "Accelerating virtual walkthrough with visual culling techniques," in *2006 International Conference on Computing Informatics*, pp. 1–5, 2006.

[22] H. Zhang and K. E. Hoff, "Fast backface culling using normal masks," in *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, (New York, NY, USA), p. 103–ff., Association for Computing Machinery, 1997.

[23] S. Kumar, D. Manocha, B. Garrett, and M. Lin, "Hierarchical back-face culling," in *7th Eurographics Workshop on Rendering*, pp. 231–240, Citeseer, 1996.

[24] C. Lee, S.-Y. Kang, K. H. Kim, and K.-I. Kim, "A new hybrid culling scheme for flight simulator," in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, pp. 1–7, Oct 2014.

[25] J. Yi, "An efficient occlusion culling algorithm of line segment intersection based on large-scale scene," in *2012 8th International Conference on Information Science and Digital Content Technology (ICIDT2012)*, vol. 1, pp. 128–130, 2012.

[26] O. Sudarsky and C. Gotsman, "Dynamic scene occlusion culling," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 1, pp. 13–29, 1999.

[27] javatpoint.com, "Computer graphics z-buffer algorithm." Available: https://www.javatpoint.com/computer-graphics-z-buffer-algorithm. accessed 29/11/2024. [Online].

[28] D. Staneker, "A first step towards occlusion culling in opensg plus," in *Proc. of the 1st OpenSG Symposium*, 2002.

[29] JEGX, "Gravitymark quick test (opengl, vulkan, direct3d12)." Available: https://www.geeks3d.com/20210719/gravitymark-quick-test-opengl-vulkan-and-direct3d12/. Accessed 03/12/2024. [Online].

[30] Z. Jia, S. Liu, S. Cheng, X. Zhao, and Z. Gongbo, "Modeling of complex geological body and computation of geomagnetic anomaly," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–10, 05 2020.

[31] K. et al., "Manifesto for agile software development." Available: https://agilemanifesto.org/. Accessed 09/12/2024. [Online].

[32] Microsoft, "First look at profiling tools (c, visual basic, c++, f)." Available: https://learn.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour?view=vs-2022. Accessed 09/12/2024. [Online].

[33] T. R. Foundation, "The r project for statistical computing." Available: https://www.r-project.org/. Accessed 09/12/2024. [Online].

[34] D. Blythe, "Rise of the graphics processor," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–778, 2008.

[35] E. Games, "Hardware and software specifications." Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/hardware-and-software-specifications-for-unreal-engine. Accessed 09/12/2024. [Online].

VI. Appendix

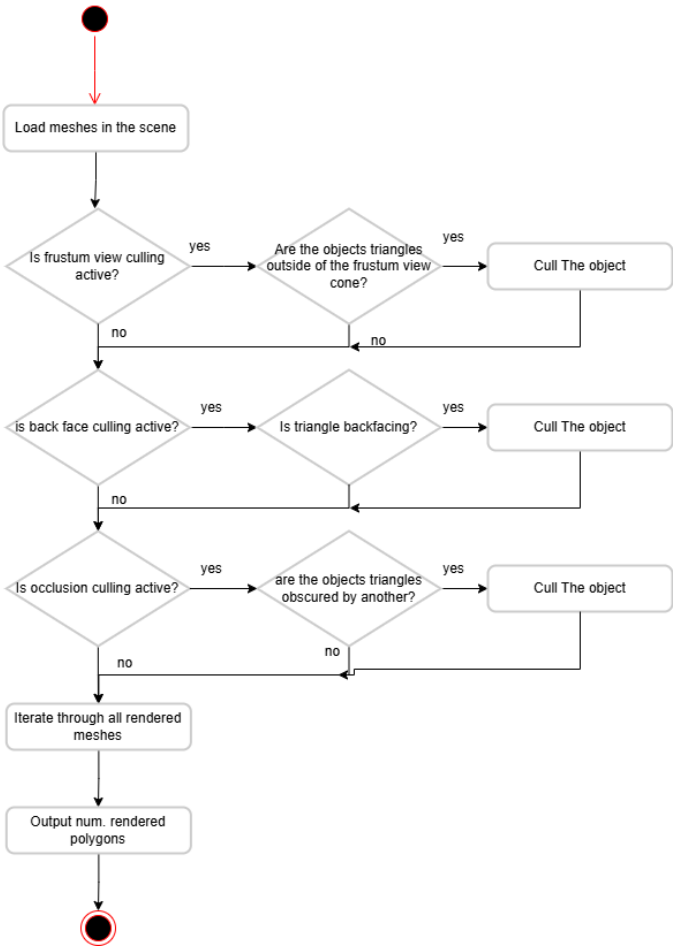

Fig. 4. The process to measure the efficiency and performance of the culling algorthms.