Real Time Graphics Environment Improvement Through Applied Culling Techniques

2204080
Games Academy
Falmouth University
Falmouth, United Kingdom

***Abstract*—As graphics simulations become increasingly complex, the performance of the hardware required to run them smoothly becomes more expensive, and inaccessible to individuals and businesses on a budget. This has created a need to run complex environments with high poly models smoothly on lower end machines. The main challenge is finding the most effective culling algorithm for the graphics environment, because some culling algorithms perform better in specific environments. There is much existing research in optimising individual culling techniques, but less research into taking a combined approach to applying culling techniques. Therefore, this paper took a combined approach to applying culling algorithms in a 3D environment. Three different culling algorithms were applied to a selection of different environments discovering that Frustum View Culling had the greatest performance improvement individually and in combination with other algorithms, but only in environments with static objects. Meanwhile, ZCulling had the words performance in smaller, densely populated environments. The paper concludes by noting the limitations of the experimental process, and suggesting future avenues for research.**

I. Introduction

Simulated 3D computer environments are an area of computing widely used in modern day life. They are required in a large variety of fields, from computer games, to medical and scientific research, and even in military simulations. However, with recent meshes and models becoming increasingly more complex and realistic, general purpose computers can sometimes struggle to maintain a consistent performance. With models comprising over hundreds of millions of polygons now becoming the norm [1], and computationally expensive techniques such as ray tracing [2] becoming increasingly common, methods of reducing the number of polygons in each model, without compromising the realism of it's appearance are needed to run graphics simulations at a consistently high frame rate. The aforementioned industries will benefit from an increase in graphics simulation performance, as they possess a need for fast and efficient three dimensional environments.
Contextual knowledge, and previous research is discussed in section II, which covers fundamental graphics programming knowledge like the graphics rendering pipeline, graphics API's and mesh generation. It also covers three types of culling techniques, and existing optimisations that have been made to them, highlighting the lack of study, and need for research into a combined approach to applying culling algorithms.
The findings of the previous research are discussed in Section III, where contextual knowledge is summarised, and a research question, and set of hypotheses are derived, which this paper strives to answer.
Section IV outlines an experiment which utilised a graphics API (Application Programming Interface) to create three interactive three-dimensional environments, in which three different culling algorithms were applied individually, and in combination with each other, to measure which combination of culling techniques performs best the environments. This helped discover which culling algorithms provide the highest increase in frame rate while culling the highest number of polys (polygons), and which environmental factors affected this the most. It also showed to what extent a combination of culling algorithms could be used to optimise different graphics environments. The method of data collection is also discussed in this section.
The results of the experiment are discussed and analysed in VIII. The data gathered is presented and visualised in this section, with statistical analysis tests also being conducted in this section.
Finally, section IX discusses implications and relevance for the data gathered, recommendations for future research, and concludes the paper.

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 [3], 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 [4].

B. Graphics APIs

An API a piece of software which acts as a contract between applications, [5] 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. [6] 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. [7] 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 [8]. On the other hand, OpenGL is a relatively higher level API, and has been the most widely adopted API [9] 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 [10]. Triangles in particular are used intensively to create these meshes [11] because of their simplicity, allowing for flexibility and efficiency [12]. 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 manipulating the polygons which make up a mesh [13]. 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 [14]. 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 [15], as shown in fig1.

D. Frustum View Culling

The first culling technique used was 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 [17]. The result is a pyramid-shaped volume projected outwards from the camera as shown in Fig2 Only the polygons inside the frustum cone are rendered [19], leading to the GPU or CPU rendering

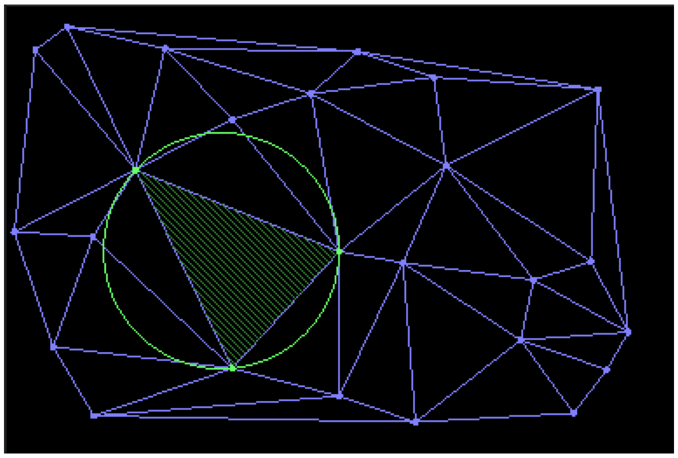


Fig. 1. An example of Delaunay triangulation used on a complex mesh. Note that the three vertices of the highlighted triangle are always on the circumference of the circle, and never inside it. [16]

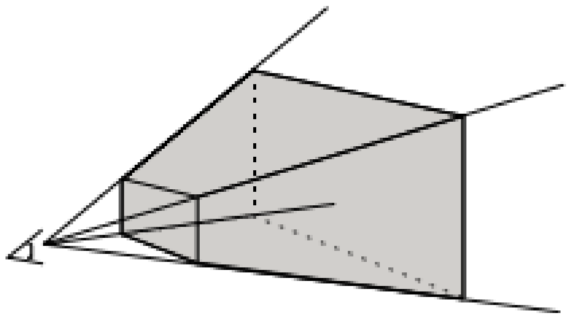


Fig. 2. The view frustum projected out from the camera view. [18].

less objects in an environment and 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 [20]. Furthermore, basic frustum view culling has been further developed and optimized by combining it with octree data structures, improving culling efficiency greatly [21]. However, the complexity of modern day environments is only increasing, resulting in un-optimized culling methods being unable to keep up [22], a clear drawback of this culling technique . This experiment aimed to negate this drawback by also implementing two other culling techniques, theoretically allowing the program to run smoother, without the use of an optimized frustum view culling technique.

E. Back Face Culling

The second culling technique used for this experiment was 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 [23]. 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. [24]. 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
        $renderBackFaces \leftarrow false$
    end if
end for
```

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 [25] 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 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% [26] - a drastic improvement.

F. Z-Culling

The final culling technique used for the experiment was Z-Culling. It works by culling objects which are a certain distance away from the camera. The intended effect is that far away objects, which the user would not ordinarily look at, are culled. A common approach to implementing Z-Culling is by utilising the depth buffer, which stores the depth values of objects in the environment [27], and checking if their distance is greater than the distance from the camera, to its far view plane. If so, it culls the object from view. This process is done at the end of the graphics render pipeline [28], one benefit to this approach is that visibility is determined regardless of the order in which objects are drawn. A drawback however, is that each pixel on the screen may be rendered multiple times - a common inefficiency [29].

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 [30]. However, OpenGL is easier to learn, and has been the more widely adopted graphics API for longer. It should be more stable, and have more widely available support.

Algorithm 2 A Z-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
        $renderCurrentObj \leftarrow true$
    end if
end for
```

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 [31], 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 a greater number of objects rendered.

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 algorithms, it can drastically decrease the number of triangles, therefore drastically increasing the performance of the program.

5) Z Culling: Z Culling makes extensive use of the depth buffer, and relies on variables such as the camera position. It will benefit from environments with larger meshes, or more sparsely populated environments, where many objects are far away from the user. The nature of the culling algorithm will also mean it benefits less in densely packed environments.

B. Hypotheses

The primary research question derived from the following research will be: "To What Extent Can the Performance of a Real Time Graphics Environment be Improved Through Combined Culling Techniques?"
Answering the primary question could also lead to a secondary research question being answered, which is "To what degree does a combined culling technique approach perform better than a singular optimised culling algorithm approach?" The experiment in this paper will not directly answer this question, however the results it produces can be compared to the results of existing research, which may lead to an indirect answer.
From these questions, the following hypotheses are derived:

1) The greater the number of culling algorithms applied simultaneously, the more diminishing the increase in frame rate per culling algorithm will be. A larger difference in performance will be observed when culling algorithms are used independently, and a smaller difference will be observed when combining culling algorithms.
2) Having three culling algorithms applied at once will provide the lowest polygon and model count, however the greater the number of culling algorithms applied, the lesser the number of polygons and models culled will decrease by.
3) The culling algorithms and combinations will be least effective in the dynamic environment. This is because only basic versions of the culling algorithms are used, and optimisations relating to object count, or dynamic scenes have not been implemented.
4) ”When applied in isolation, back-face culling will improve performance the most” This is working off the assumption that back-face culling should almost always be culling approximately half of all triangles in the environment.
5) ”When applied in isolation, Z culling will improve performance the least on average. This is working off the assumption that Z culling is dependent on both the camera location and the direction in which it is looking, and therefore will not be culling the optimal number of polygons all the time

IV. Computing Artefact

A. Description

The artefact created for this experiment was a series of three dimensional OpenGL environments. The user could change the chosen environment before running the code, with the three choices being ”DENSE” , ”SPARSE”, and ”DYNAMIC”. These names were stored as an enum variable and could be easily changed in the code. Each environment contained the following:

- A matrix of high poly sphere models. The models were affected by the culling algorithms used. Statistics such as frame rate, GPU & CPU performance were affected by these meshes, and differed slightly depending on the environment. Each model contained 12288 polygons, and were laid out in a 25x25x25 matrix, leading to a default number of 15625 models, and 192 million polygons in the scene. The layout and behaviours of these spheres varied depending on the chosen environment. In the ”DENSE” environment, the spheres were packed closely together. In the ”SPARSE” environment, the spheres were significantly further apart than the dense environment, but maintained the same size and behaviours. In the ”DYANMIC” environment, the spheres were placed a little further apart from each other than the ”DENSE” environment, but much closer together than the ”SPARSE” environment. The most notable difference was that all the spheres rotated around their own individual centre points. An example of the environment is show in figure 3
- An interactable camera. This was the users primary method of interaction and navigation in the environment. It contained the code for keyboard movement, mouse movement ,zoom, and field of view.

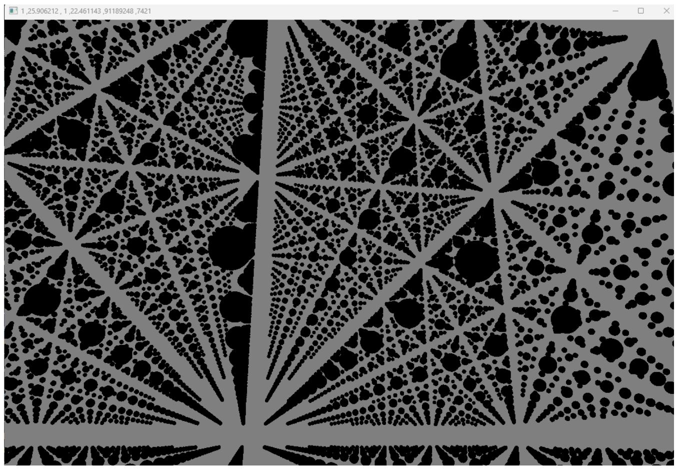


Fig. 3. The artefact ”SPARSE” environment

The artefact allowed the user to change the environment in the artefact source code. There was a choice of three enum variables, which each represent the ”DENSE”, ”SPARSE”, and ”DYNAMIC” environments, which could be easily changed. Depending on this value, when run, the artefact rendered the aforementioned matrix differently - the models were be close together in the ”DENSE” environment, further away from each other in the ”SPARSE” environment, and moving in the ”DYNAMIC” environment. Relevant data such as the iteration count, frame rate, model count, polygon count, and active culling algorithm were shown in the top left, as the software window name.

This experiment was 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 avoided inaccuracies created by hardware requirements, and allowed for the sole focus of the experiment to be on the algorithms used, and the environments they were used in. The following project’s repository can also be found here: [1]

B. Development Methodology

This artefact made great use of version control, with branches being used to develop each culling algorithm, and each environment. This ensured that each culling algorithm was tested individually before being added to the main project, helping to simplify future unit testing by ensuring each algorithm worked in isolation before being pushed to the main branch. To avoid over-scoping, the initial scope of the project was set low, with the artefact being built upon iteratively, and features being added one at a time. The experiment process also made use of Agile software development methodology,

[1]the repository is available at: https://github.falmouth.ac.uk/GA-Undergrad-Student-Work-24-25/COMP302-WA278394-2204080.git

which advocates for a reflective and adaptive development process [32]. This was ideal for an experimental piece of software, because the development of the artefact remained flexible. The iterative approach previously mentioned is advocated for in the Agile methodology, an ideal approach because features could be added and removed to fit the scope of the project.

C. Quality Assurance

1) Construct Validity: To ensure high quality research, this experiment measured the mean values for frame rate, memory usage, CPU usage, and GPU usage when using the applied culling algorithms. The only user input in the environment was rotation around the matrix of models, ensuring the entire environment was viewed and culled. Microsoft Visual Studio's profiling tool [33] was used to gather CPU usage, GPU usage and memory usage, while frame rate was recorded and gathered via code, because the profiling tool did not do this automatically. These variables were exported to a CSV, and analysed using R code [34].

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 modelling software and even computer-aided content creation and medical screening [35]. Games and Modelling 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 [36]. 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 frame rate, 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 maintained 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 were pre-rendered, which allows for greater specificity and variety for testing the culling algorithms in. Utilizing the same culling algorithms in the same scenes should therefore give very similar, if not identical results each time the experiment is run.

4) reproducibility : This experiment, if carried out via the same method, should be very reproducible. The culling algorithms used had no optimizations or alterations made to them, with the novelty of the project being in applying multiple culling algorithms at once. Therefore, if the same culling algorithms are applied, and assuming the number of models in the environment are large enough, this experiment should be replicable.
This experiment could also be applied to proprietary game engines such as Unity, and the aforementioned Unreal Engine, which also allow users to apply various culling techniques. The results may differ however, because both engines may incorporate other factors into their culling algorithms, instead of the OpenGL/C++ algorithm which this experiment will use. As a result, this may result in a type 1 (false positive) error, as the algorithm may not increase performance as much as expected.
The largest place for error in the reproducibility of this experiment however, lies in the machine in which it runs on. The machine used to run this experiment was a mid-high end machine (RTX 3060, 32GB RAM and a 2.3GHz clock speed.) While the triangle count will remain the same across all devices, performance metrics such as mean frame rate would be less noticeable on high end machines, because they won't struggle with the effects of multiple high poly meshes to begin with.

5) unit tests: The following parts of the artefact were unit tested:

- Culling Algorithm Functions - The unit tests in figure 19 show each of the three culling algorithms working independently. This is necessary because the culling algorithms were an independent variable in the experiment, with the results being reliant on a correctly functioning culling algorithm.
- Environment Drawing Functions - The unit tests in figure 19 show each of the three environments being fully rendered. Environment choice was the other independent variable in this experiment, with the results varying depending on the chosen environment.

V. Methodology

A. Research Philosophy

From an ontological and epistemological point of view, this paper assumed that the statistics displayed and gathered from the experiment are the truth based on observation and evidence, which is what constitutes valid knowledge. This experiment took a pragmatic and objective approach to research. The optimization of of graphics simulations in general could have wide reaching benefits to society in multiple disciplines - as mentioned in the previous subsection such as military and medical fields. Objectively, the values of this experiment will not change due to human perception or opinion.

B. Research Method

1) Sampling: The data from each iteration was taken during the programs runtime, and recorded 30 times per second. The program will run for 30 seconds, of which data will only be collected after 15 seconds have passed. This is to prevent unstable and anomalous results which may occur during the programs start up caused by the initial environment rendering. This means that an average of 900 values for frame rate, model count, and polygon count will be recorded for each iteration, and stored in the results. 38 iterations for each scene will be sampled, resulting in 114 samples taken per culling algorithm and combination.

To establish a baseline of values, 114 samples (again 38 per scene) will be taken where no culling algorithms are active. This establishes a point in which the values can be compared against, and where improvements can be measured from. The culling algorithms used in this experiment were tested individually, pairwise, all at once, and with none applied, totalling up to 24 (8 total different configurations x 3 different environments) different conditions. During these tests, the following data was automatically recorded and written to a list, where it was then all written to a CSV in one go upon the programs termination:

- frame rate - This was used to measure the performance of the program before and after the culling algorithms are applied, with a higher frame rate being more favourable than a lower frame rate. This statistic was collected through code. By utilising the glfw library's [37] "get time" function, and the time between the current frame and the last frame. The values were written to a list every 1/30th of a second, where upon the programs termination, the average was found, and written to the CSV file for that specific iteration.
- Model count - This statistic measured the number of models currently rendered in the scene. This value was incremented every time the function for drawing a model was successfully called, and decremented every time a model had 0 rendered polygons on screen. This value was updated every frame.
- Polygon count - This statistic measured the number of polygons currently rendered on the screen. This value was calculated by taking the model count and multiplying it by the number of polygons per model, which was a predetermined value, calculated by loading the model into Blender modeling software [38], which gives an accurate polygon count of models.

2) Measurement: To measure the ultimate effectiveness of the culling algorithm in each environmnent (defined as "EScore", an overall efficiency score will be calculated based on the percentage frame rate improvement, and the number of remaining polygons:

$$EScore = \frac{\%framerateImprovement}{\%PolygonReduction}$$

This equation will measure the ratio of frame rate improvement to polygon count for each culling algorithm combination. A higher EScore value indicates that a culling algorithm, or combination are better at improving the performance and culling the polygons in that chosen environment.

To get the percentage frame rate improvement, the average frame rate of the selected culling algorithm in its environment will be divided by the average baseline frame rate in the selected environment, and multiplied by 100:

$$\%framerateImprovement = (\frac{avg.FPS}{avg.BaselineFPS}) \times 100$$

This equation gets the percentage increase in frames per second, when comparing two average frame rates in the same environment, with a higher percentage increase suggesting a greater performance increase by using the selected culling algorithm.

To get the percentage polygon reduction, the average number of rendered polygons with the culling algorithm enable will be divided by the average number of rendered polygons in the baseline tests, which will also be the total number of polygons rendered. The total will be multiplied by 100 to get a percentage:

$$\%PolygonReduction = (\frac{avg.Polys}{avg.BaselinePolys}) \times 100$$

3) Experimental Design:

C. Variables & Libraries

1) Variables: This experiment contains multiple independent and dependent variables, and as such will utilise a factorial experimental design, which will allow for multiple factors to be researched simultaneously.

- algorithm : bool - Independent variable, the algorithm was changed frequently during the testing process to gather which one could cull the most polygons while keeping GPU and RAM usage lower. These variables were gathered and coded as boolean values, with 0 being inactive, and 1 being active.
- environment : enum - Independent/Controlled variable, the environment in which the chosen algorithm will be applied in. These varied to see if certain culling algorithms perform better in specific environments. An enum was used to make changing the environment via code easier.
- model count: float - Dependent variable, the number of models rendered on the GPU. This was used to measure how effective the culling algorithm is during runtime. A higher number of models culled meant higher effectiveness.
- polygon count: float - Dependent variable, the number of polygons rendered on the GPU. This was used to measure how effective the culling algorithm was during runtime, with a higher number of polygons culled meaning a higher effectiveness. This variable is derived directly from the model count.
- frame rate: int - Dependent variable, number of frames displayed per second. This was used to measure the performance of each culling algorithm. A higher frame rate meant a better performance.
- numpolygons: int - Controlled variable, the number of polygons per model. This was used to determine the polygon count variable, and could be adjusted, allowing for different models to be used in the experiment.

2) Libraries:

- GLAD [39] - An open source library which manages OpenGL's function calls to be operating system and hardware independant. It was used to shorten and streamline the coding process siginifcantly. Furthermore, the location of many OpenGL specific functions are not known at run-time, due to different versions of OpenGL's drivers being required for different hardware. GLAD helps by querying this during compile time, rather than runtime.
- GLFW [37] - A C Library written specifically for OpenGL. It allows for defining window parameters and handling user input, both of which are essential for the user to navigate a 3D environment, which the experiment requires.
- GLM [40] - A header only C++ mathematics library, made specifically with graphics software in mind. It allows for easy use and calling of maths functions such as number randomisers, Vector maths, matrices, & Quaternion mathematics. These features were necessary for this project, as vector maths and matrices are commonly used in computer graphics. The project made extensive use of this library.
- Assimp [41] - A library which allows for loading and rendering of 3D file formats. This was used to import the sphere models used in each environment. This saved having to individually draw each triangle for each model, which would have been significantly less efficient.

VI. Ethical Considerations

The research carried out is considered a low-risk experiment, because no human participation is involved, since all statistics gathered are from a piece of software. Ethical approval for this project has been obtained. Furthermore, the testing of this artefact will only be performed on a personal machine, therefore posing no threat to any publicly used/owned devices, or any other devices connected to a network. Furthermore, this experiment, and the simulations created will not be used outside of purposes for this experiment.

The nature of graphics simulations in computing have very widespread uses however, and incorrect use of culling algorithms could have a large number of misuse cases. On the low severity end, this includes fields such as Entertainment and 3D modelling & architecture, in which the consequence of incorrectly culling a model would be a detraction from the user enjoyment. On the more severe end of the spectrum involve areas such as healthcare, aerospace defense and scientific research, where an incorrectly culled model could result in simulations of life-threatening and dangerous objects, such as cancers, military equipment, or hazardous material being removed from view, and therefore missed in a simulation, leading to large amounts of damage and potential loss of human life. In military and medical fields where this may apply, the software's development would be required to follow medical simulation, and military codes of ethics and conduct [42] [43]. Conversely, a poorly optimised culling could lead to a severe drop in frame rate and quality. This could lead to a slow simulation, which could be dangerous in a real-time context where the previously mentioned fields are invovled. Culling algorithms are therefore more suitable than not in these industries for this reason, as they can assist in making software and simulations run faster. However they may not be entirely suitable in situations where high levels of detail are essential. While the issue of performance could be solved with higher end machinery in this case, this would not be possible without appropriate funding and purchasing ability of such hardware, leading to culling algorithms being suitable in such a scenario.

VII. Data Management Plan

A. Data Gathering

To gather the experimental data required to answer the research question and hypotheses, the artefact wrote the current iteration, environment, average frame rate, average polygon count, and average model count to a list during runtime. Upon the programs termination, the averages of the frame rate, model count, and polygon count for that iteration were calculated, and appended to a CSV file specific for that combination of algorithms. This kept the performance decrease of writing to a large CSV file to a minimum, and pre-calculated average values, shortening the length of the CSV files. Separate CSV files for each algorithm combination and environment keep the results organised, and allow for easier data analysis. The table below I shows an example of how the CSV files appeared.

Iteration	avg.fps	avg.model	avg.poly	env
1	15.223	15625	1.92e+8	DENSE
2	14.927	15625	1.92e+8	DENSE
3	15.029	15625	1.92e+8	DENSE

TABLE I
Example Layout of a Dataset

A total of 24 CSV files were created containing the relevant data, each having 38 iterations, each containing the average FPS, model count, polygon count, and chosen environment for that scene.

B. Data Analysis

Once collected, the data underwent a linear regression statistical significance test using R. To allow for this, each dataset was appended to include a binary, correlating to the activity of each of the culling algorithms. This allowed for individual and combinations of algorithms to be represented, and related to their relevant stats. Each dataset would be read and assigned to a variable, which would be used to merge all the data into one large data frame. The data frame would be cleaned, and then have the linear regression statistical signifcance test run using it. The ideal correlation coefficient (r) value would be either a strong positive (+1) or strong negative (-1) value, with the ideal p-value being 0.05 or less.

For answering each of the hypotheses, the larger data frame was used, with appropriate columns of data being selected, compared, and graphed.

C. Data Display

For hypotheses 1 and 2, a point and line graph were used to compare the average frame rate, average model count, and average polygon count against the number of active culling algorithms. This required temporarily appending a new variable to the dataframe, which kept track of the number of currently active culling algorithms. A plot and line graph would show any correlations between the measured statistics, and the number of culling algorithms active, therefore providing answers to these hypotheses. For hypothesis 3, a scatter graph was used, which compared the average FPS against the average polygon counts for each environment. This graph could also be used to draw correlations and conclusions from, to answer hypothesis 3. For hypothesis 4, the frame rate would be compared with the environment, but only when a singular culling algorithm was used. This graph would be required to display the average fps of the baseline environments, and when the culling algorithm was active. This was more appropriate to show in three different graphs, each of which were box plots. This allowed hypothesis 4 to be answered.

Finally, to determine the EScore of each culling algorithm, the previously defined equations in sectionV were implemented. To get a data set with only the baseline values for frame rate and polygon count, a sub set was created, which only displayed data where all three binaries for the culling algorithms were set to 0. The average baseline values for frame rate and polygon count were then taken from this new sub set. Using the original data set, the average frame rates and polygon counts for each culling algorithm combination were also gathered. Finally, the two averages were put into the previously defined equations and assigned an EScore value.

VIII. Hypothesis Testing

A. Statistical Significance Testing

This experiment was carried out under the assumption that a one way ANOVA test would be used to find its statistical significance. Figures 13 and 14 in the appendix display that a sample size of 112 was required to find significant data. To ensure an equal number of iterations for each environment and culling algorithm combination, 114 samples were taken, having 38 for each environment, for each culling algorithm combination. The culling algorithms applied, and environments they were in were highly varied, which justified a larger effect size of 0.4.

During the projects development however, using Linear Multiple Regression as a method of finding statistical significance became more favourable, because ANOVA did not account for the combinations of culling algorithms. Figure 15 shows the G*Power tests for this stats test method, which returned a sample size of 77 with a power of 0.8. The original sample size of 112 was adhered to while collecting data, which a post hoc test in Figures 17 & 18 returned a power of 0.941, therefore decreasing the probability of a type 2 error by 0.141.

B. Results and Findings

```
Multiple R-squared:  0.5667,
Adjusted R-squared:  0.5634
F-statistic: 169.1 on 7 and 905 DF,
p-value: < 2.2e-16
```

The code excerpt above shows the results of the linear regression statistics test run on the data. The model shows that approximately 56% of the variance is explained by the variables in the experiment, and that the complexity of the model also accounts for roughly 56% of the variance. The F-statistic is very high, and the p-value extremely low, therefore suggesting that at least one variable in these results is highly statistically significant, and that the null hypothesis can be strongly rejected. Finally, a p-value of 2.2e-16 is very low, and below 0.05. Therefore, the results observed are highly unlikely to be random, supporting the relationship between the variables.

1) hypothesis 1 - A greater number of culling algorithms applied simultaneously will yield diminishing returns in average frame rate per culling algorithm.: Figure 4 shows a positive correlation between frame rate, and the number of active culling algorithms. When 0 culling algorithms were active, the average frame rate sat in between 12.5 - 15 frames per second, at approximately 13.75 frames per second. When only 1 culling algorithm was active, the average frame rate increased slightly to approximately 18.25 frames per second, showing a 32.7% increase. With 2 active culling algorithms, the average frames per second increased from 18.25, to around 24 frames per second, showing a 31% increase in frame rate in between the two, Finally, with all three culling algorithms active, the frame rate averaged at 34 frames per second across all three environments, for a 41.6% increase in frame rate in between the two algorithms. Overall, across all three environments, there was on average, a 147% increase in frame rate between having no culling algorithm applied, and having three culling algorithms applied overall.

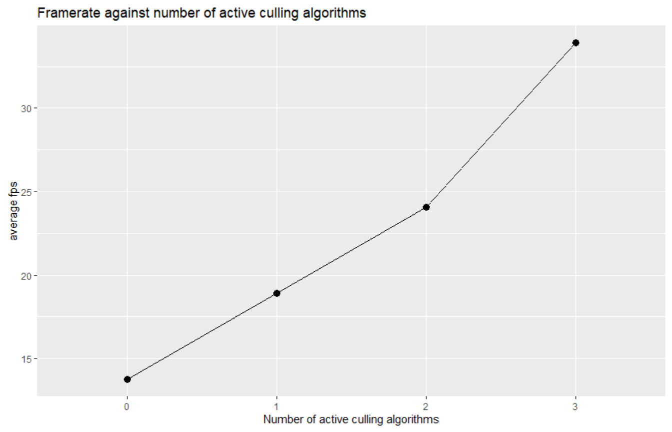


Fig. 4. Frame rate against the number of active culling algorithms

The results from figures 4 and 20 refute this hypothesis. There is a 29.87% increase in frame rate between 0 and

1 active culling algorithms, and a 27.28% increase between 1 and two active culling algorithms, which would initially appear to weakly support the hypothesis. This is followed by a much larger 40.96% increase in frame rate for all three culling algorithms, which conversely refutes the hypothesis more strongly. This implies that applying a greater number of culling algorithms to an environment will show greater increases in frame rate for each culling algorithm applied.

A limitation of these results is that it does not account for the difference in performance of each culling algorithm. Some culling algorithms will inherently increase performance better than others, meaning the hypothesis could be supported or refuted depending on the combination of culling algorithms used. Another potential limitation, is the low number of total culling algorithms used. While three culling algorithms gives a rough idea of the correlation between the two values, a greater number of culling algorithms applied may give a more accurate answer.

2) Hypothesis 2 - All three culling algorithms being applied will provide the lowest polygon count, but also a lower decrease in polygons per culling algorithm active.: Figures 6 and 5 show a negative correlation between the average model and polygon counts against the number of active culling algorithms. With 0 culling algorithms active, the average number of polygons was 192 million (192000000), and the number of active models was 15625. With 1 culling algorithm active, the average number of rendered polygons decreased to approximately 111 million, and the number of models dropped to 11690 models. This was roughly a 42% decrease in polygons, and a 25% decrease in models. With 2 active culling algorithms, the polygon count decreased to 60.9 million (6.09e+07), and the model count decreased to approximately 8166 models, showing a 45.14% decrease in polygons, and a 30.12% decrease in models. Finally, with all three culling techniques active, the number of polygons was reduced to approximately 29.7 million (2.97e+07) polygons, and the number of models was decreased to 4848. This is a 51.23% decrease, and a 40.63% decrease respectively between the two statistics. Overall, across all three environments, a 84.53% decrease in polygons, and a 68.97% decrease in models was observed.

The results appear to support this hypothesis, because the lowest polygon and model counts are observed with all three culling algorithms active. Furthermore, the polygon count decreases the most with just one culling algorithm active, less so with two, and even less so with three.

A limitation of these results however, is the limited number of culling algorithms used. The correlation of the results suggest that the average number of rendered models will continue to decrease at an almost constant rate. This does not line up with the average number of rendered polygons, which has less of a percentage decrease the more active culling algorithms there are. Since models and polygons are mutually exclusive, it could be assumed that a decrease in percentage decreaase should be present for the number of rendered models too, however this is not displayed by the graph.

3) Hypothesis 3 - The culling algorithms will be least effective in the dynamic environment due to the unoptimised nature of the culling algorithms: Figure 7 shows various correlations for the average frame rates, against the average polygon counts in each environment, and therefore how each culling algorithm is affected by the environment itself. For the ”DENSE” environment, there are two clusters of values. The first shows a cluster between an average polygon count of 5.0e+07, and 1.0e+08, showing average frame rates between 20-45 frames per second. The second cluster is grouped between an average polygon count of 1.25e+08 and 1.92e+08 rendered polygons, and shows an average frame rate roughly between 21-13 frames per second. The two clusters form a weak negative correlation between average frames per second and the average number of rendered polygons.

The ”DYNAMIC” environment forms three smaller clusters. The largest cluster ranges between roughly 1.25e+07 and 6.25e+07 rendered polygons on average, displaying an average frame rate between 4-6 frames per second. The second cluster varies between roughly 6.25e+07 polygons, and 9.0e+07 polygons, at a consistent frame rate between 17-18 frames per second. The third group varies between an average rendered polygon count of roughly 1.65e+08, and 1.8e+08 rendered polygons, at a similar frame rate between 17-18 frames per second. The three clusters form a weak positive correlation between the average frame rate, and the average number of rendered polygons.

The ”SPARSE” environment formed several clusters on the graph. The first cluster situated in the top left of the graph, presents an average polygon count between 4.0e+06 and 5.0e+06, with an average frame rate between 55 and 68 frames per second. The second cluster forms between 4.0e+07 and 1.13e+08 average rendered polygons, and between 39-55 frames per second. The third cluster formed between 2.6e+07 and 9.6e+08 average rendered polygons, with a consistent frame rate between 17-18 frames per second. A fourth and

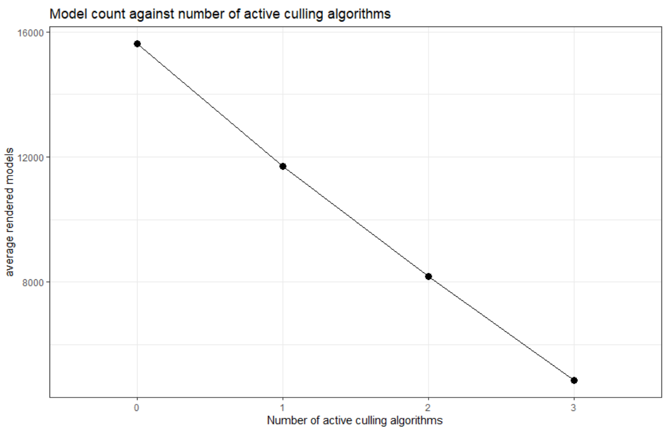


Fig. 5. Model count against number of active culling algorithms

final cluster forms at 1.92e+08 rendered polygons, with an average frame rate between 14-15 frames per second. All four clusters form a weak negative correlation between average frames per second, and the number of rendered polygons, however there is a larer confidence interval than with the other two environments.

The data from figure 7 appears to support this hypothesis. The graph shows a larger grouping of data with the lowest frame rate in the DYNAMIC environment. Furthermore, the frame rates appear to increase, with the decrease in rendered polygons in the DENSE and SPARSE environments, with the greatest effect being in the sparse environment. This suggests that culling algorithms work best in sparsely populated environments, where objects are rendered far away from the user, and work the worst in environments with lots of dynamic, moving objects. Culling algorithms appear to be effective in densely populated environments, where many objects are rendered closer to the user, but less so than in sparsely populated ones.

Although the results do not explicitly show this, a potential reason for the poor performance in the DYNAMIC environment could be the unoptimised nature of the culling algorithms, particularly with frustum view culling. This particular frustum culling algorithm was not optimised to work hierarchically, or with dynamic objects, meaning that every frame, the algorithm would check the entire view matrix, which in this case, would contain tens of thousands of models, check each one to see if it has moved, and then check if it was outside the view cone, and cull it if so. This is an incredibly inefficient process with O(n^2) time complexity.

The high performance in the SPARSE environment could be attributed to the nature of frustum view culling. In this environment, many objects would regularly be behind the camera and therefore culled. When combined with Z-Culling too, it would also cull many distant objects too, further increasing the frame rate. This would also justify the weaker performance increase in the DENSE environment, because there would be less objects culled via frustum culling and z culling.

The unoptimised frustum culling algorithm in this case, is a big limitation for this data. It does not accurately represent the potential performance increase that culling algorithms could have in a dynamic environment. Furthermore, it does not accurately represent culling algorithms in combination with frustum culling because of this lack of optimisation.

4) Hypothesis 4 - Back face culling will improve performance the most when applied in isolation due to it always culling approximately at least half of all polygons: Figures 8, 9 and 10 show the performance of each culling algorithm applied individually to each environment. Figure 8 shows that the mean frame rate increased from a value between 15-16 frames per second, to a mean frame rate of 20 frames per second when back face culling was applied in the DENSE environment, with an upper bound of around 26 frames per second, and a lower bound of 14-15 frames per second. In the DYNAMIC environment, the median frame rate was around 12-13 frames per second, while the baseline was around 7-8 frames per second. The upper bound however was slightly lower at around 16 frames per second, compared to the baseline upper bound which was between 16.5-17 frames per second. The lower bounds were roughly the same, at about 6-7 frames per second. In the SPARSE environment, the median frame rates for back face culling were relatively similar, with the baseline being around 17-18 frames per second, and the applied mean frame rate being around 18-19 frames per second. The upper bounds for the baseline were around 51 frames per second, and the lower bounds being around 13 frames per second. In contrast, the upper bounds with the culling algorithm applied was around 55 frames per second, and the lower bounds being about 17-18 frames per second.

Figure 9 shows that in the DENSE environment, when frustum view culling was not active, the mean frame rate was around 15 frames per second, the upper bounds at around 16 frames per second, and the lower bounds at around 14 frames per second. While the culling algorithm was active, the mean frame rate increased drastically to 21 frames per

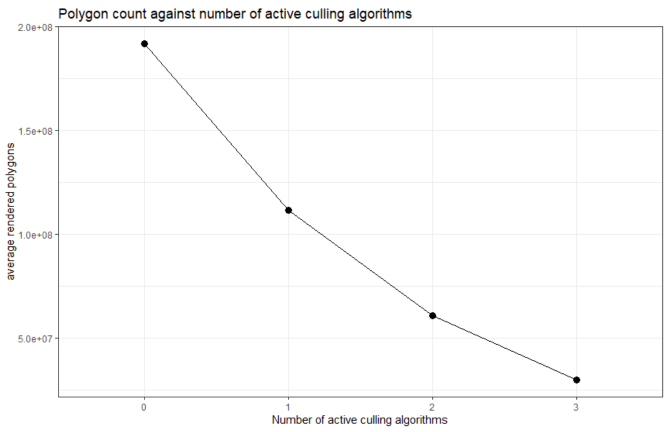


Fig. 6. Polygon count against number of active culling algorithms

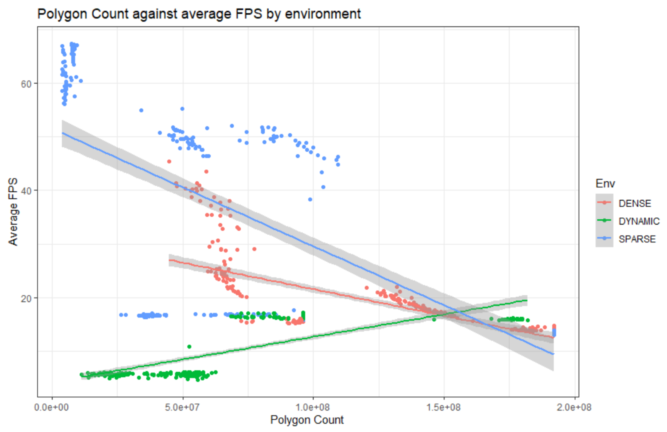


Fig. 7. Average rendered polygon count in each environment

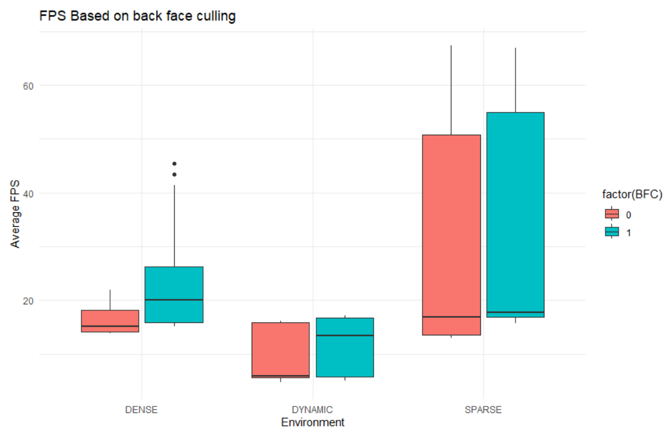


Fig. 8. FPS based on back face culling

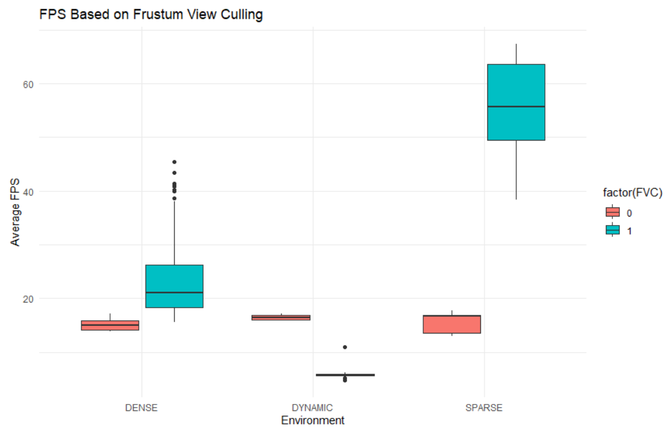


Fig. 9. FPS based on frustum view culling

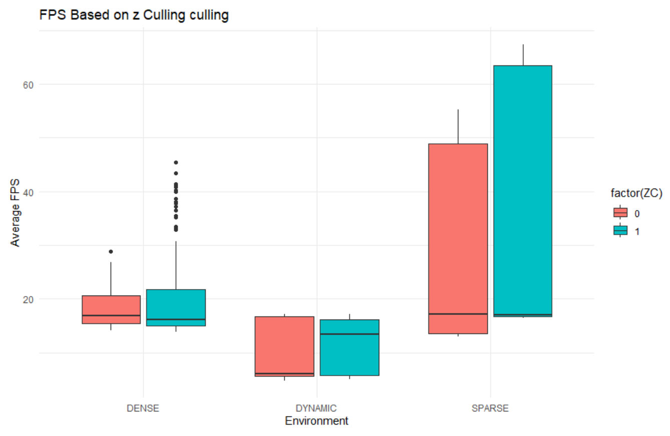


Fig. 10. FPS based on Z Culling

second, the lower bounds to around 19 frames per second, and the upper bounds to 16 frames per second. In the DYNAMIC environment, the baseline mean frame rate sat at around 16 frames per second, with the upper and lower bounds being roughly half a frame above and below that respectively. While the culling algorithm was active, the mean, upper, and lower bounds all sat closely within around 5 frames per second. Finally, in the SPARSE environment, the baseline median frame rate sat at around 17 frames per second, with the lower bound being around 14 frames per second and the upper bound also being at around 17 frames per second. With the culling algorithm active, the median frame rate increased greatly to around 56 frames per second, the lower bound being around 49 frames per second, and the upper bound being around 63 frames per second.

Figure 10 shows that in the DENSE environment, when z culling was not active, the mean frame rate was around 17 frames per second, the upper bound was 21 frames per second, and the lower bound was around 15 frames per second. In comparison, when the culling algorithm was active, the mean frame rate was slightly lower at around 16 frames per second, and the lower bound was slightly lower at around 14.5 frames per second. The upper bound however was higher at around 22 frames per second. There were however, many outliers above the maximum. In the DYNAMIC environment, when the algorithm was not active, the mean frame rate was around 5 frames per second, the lower bound was between 4-5 frames per second, and the upper bound was around 16 frames per second. With the algorithm active, the mean frame rate increased to around 13 frames per second, with the lower bound similarly being around 5 frames per second, and the upper bound being a slightly lower value between 15-16 frames per second. In the SPARSE environment, without the culling algorithm active, the mean frame rate was at around 16 frames per second, with the lower bound being aroud 13 frames per second, and the upper bound being 49 frames per second. With the algorithm active, the mean frame rate stayed roughly the same, with the lower bound increasing to a value between 15-16 frames per second, and the upper bound largely increasing to 63 frames per second.

The data collected refutes this claim, because the largest increase in frame rate came from frustum view culling being applied, rather than back face culling. A possible reason for this could be because even though back face culling is always active, every model is still rendered on the screen, whereas with frustum culling, some models are completely culled, requiring the graphics render pipeline to do less work. Back face culling had only a slightly lower mean frame rate than when frustum culling was used in a dense environment. This may suggest that in dense environments, frustum culling may be so limited, that back face culling becomes a probable alternative.

5) Hypothesis 5 - Z culling will improve performance the least when applied in isolation due to its dependency on camera position and rotation.: The data collected supports this claim. In the cases of the DENSE and SPARSE environment, the mean frame rate decreased while the culling algorithm was active. One reason why could be due to a potential bottleneck in the algorithm, in which the time it takes to cull objects outside of the camera's view outweigh the potential benefits. This could be feasible in a sparsely populated environment, however in a densely packed environment, another potential reason could be because the algorithm isn't able to cull polygons. If all the objects in the dense environment are too close to the camera view, then the algorithm would not cull the objects, yet still be running the code. This would justify the slight decrease in mean frame rate observed in the DENSE environment.

A limitation of this data is that it does not appear to correctly represent Z Culling and its potential. The data suggests that Z Culling would generally only increase the upper bounds and maximum values of which the frame rate could be. A further limitation shows in the DYNAMIC environment. the mean value for frame rate without the culling algorithms active was always very low. Z Culling brings it to a level comparable to that of the other environments, but does not improve it any further. This suggests that Z Culling may only bring performance to a set level, instead of increasing frame rate as much as possible.

Figure 11 shows the EScores for each culling algorithm and their combinations. In descending order they are:

- All Combined - 15.02 (E.ALL)
- Back Face Culling + Frustum View Culling - 7.14 (E.BFCFVC)
- Frustum View Culling + Z Culling - 6.06 (E.FVCZC)
- Frustum View Culling - 3.402 (E.FVC)
- Back Face Culling + Z Culling - 3.08 (E.BFCZC)
- Back Face Culling - 2.29 (E.BFC)
- Z Culling - 1.43 (E.ZC)
- No Culling (E.Baseline)

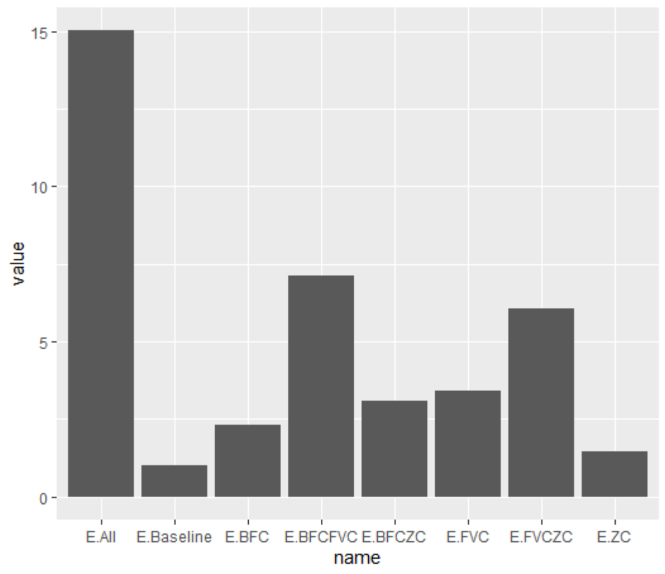


Fig. 11. Efficiency Scores of each culling algorithm combination

This data supports hypothesis 5, as Z Culling has the lowest EScore out of the three individual culling algorithms. The data also refutes hypothesis 4, as Back Face Culling does not have the highest EScore out of the individual culling algorithms, but rather Frustum View Culling instead.

From this data, we can also analyse the efficiency of each of the culling algorithms. No culling served as the baseline value of 1.0, and all three algorithms combined served as the upper limit of 15.02, with all the other culling algorithm combinations lying somewhere in between. Frustum View Culling provided the highest EScores, being present in all four of the top performing algorithms. This would suggest that frustum culling is a versatile and powerful culling algorithm, which can be combined with other algorithms to optimise environments to a much greater extent. The EScore of all culling algorithms applied was almost double that of the second largest EScore given by Back face and Frustum culling. This could be further evidence refuting hypothesis 1, because it implies that with a greater number of culling algorithms applied, the EScore increases exponentially rather than linearly.

IX. Discussion

A. Research Summary

Modern graphics simulations and games are demanding more powerful and expensive hardware requirements, making them less accessible to their target audience who are unable to keep up with the demand as software makes use of increasingly complex models and environments. One method of allowing users with lower end hardware to run these applications is through optimisation via culling algorithms. Optimised culling algorithms are highly complex, and take a significant amount of time to create.

This paper proposes a combined culling algorithm approach to optimising a 3D environment with a large number of complex models and polygons, to increase the frame rate and decrease the amount of visibly rendered geometry. By comparing the performance of different environments, with a number of culling algorithm combinations applied, the paper explores to what extent these environments can be optimised by using a combined approach. the paper also calculates the efficiency of each combination by assigning them an "EScore", which helps determine to what extent each combination is capable of increasing the frame rate of each type of environment.

The data gathered displayed a positive correlation between the average frame rate of the environment, and the number of culling algorithms simultaneously applied. It also displayed that increasing the number of active culling algorithms from 2 to 3 yielded a greater increase in frame rate, than from increasing the number of active culling algorithms from 1 to 2. This evidence would support the idea that culling algorithms applied in combination could optimise the performance of a graphics environment to a considerable extent.

Further data gathered showed a negative correlation between the average model and polygon count against the number of culling algorithms simultaneously applied. It showed that a greater number of applied culling algorithms will lead to a smaller number of visibly rendered models and polygons. The rate at which the number of visible models decreased somewhat consistently, with there being a slightly greater decrease between 0 and 1 culling algorithms being applied, and 1 & 2, and 2 & 3 culling algorithms being applied. When the polygon count was measured, a much greater decrease was observed between 0 and 1 culling algorithms being applied, than with 1 & 2, and 2 & 3 culling algorithms. The rate of decrease in polygon count was lowest when going from 2 applied culling algorithms to 3. This evidence would also support the idea that culling algorithms applied in combination could optimise the performance of a graphics environment. However, it suggested that there is a limit to the number of polygons and models which could be culled before diminishing returns. Therefore, it does not support this view to as strong an extent as hypothesis 1.

Data measuring the average polygon count in each environment against their average frame rate values was also gathered. In sparsely populated environments, there is a negative correlation between the number of visible polygons and the average frame rate, with higher frame correlating to lower polygon counts. In densely populated environments, there was a weak negative correlation between the number of visible polygons and the frame rate. This is likely because two of the three culling algorithms (back face and z culling) are reliant on objects being outside of the camera view cone. This is less likely to happen in objects are closely packed together. In dynamic environments with moving objects, there was a weak positive correlation between the number of visible polygons and the average frame rate. Overall, the average frame rates were lowest in the dynamic environment, and highest in the sparsely populated environment. This data suggested that a combined algorithm approach can improve the performance of a graphics environment to an excellent extent in sparsely populated environments, a considerable extent in densely populated environments, and a lesser extent in dynamic environments. This aligns with the knowledge outline in the literature review, which stated that a common flaw in frustum culling algorithms is their inability to keep up with more complex environments.

For measuring the performance of individual culling algorithms, the data gathered refutes hypothesis 4 and supports hypothesis 5. This therefore proposes that back face culling can only optimise a graphics environment to a lesser extent, and that Z culling would improve performance to an even lesser extent. The results show that frustum view culling would optimise a graphics environment to the greatest extent.

These results are supported by the EScores gathered for each culling algorithm combination, where algorithm combinations including frustum view culling had the highest values. This suggests that frustum culling will optimise an environment to a greater extent than back face and z culling.

B. Limitations

The main limitation of this experiment lies in the complexity of its design. The research question aiming to be answered is relatively vague, leading to an experiment being created which answers the question in a roundabout way. Furthermore, the experiment contains two independent variables, making the experiment exponentially more complex to build upon. There are 8 (2^3) algorithm combinations in 3 environments, leading to 24 possible configurations. Adding one more culling algorithm to test leads to 16 (2^4) algorithm combinations, and therefore 48 configurations. This limitation resulted in a lower number of culling algorithms being tested, and makes future work on this experiment exponentially harder.

Another limitation of this experiment lies in the environments used in the experiment. Each environment contained a matrix of sphere models to test the culling algorithms on. A pre-determined number of models are rendered during compile time, therefore having no effect on the frame rate. Therefore the experiment does not take into account, a scenario where new models are being generated during runtime, the effects that may have on the frame rate, and whether or not culling algorithms are an appropriate method of optimising such an environment.

C. Conclusion

The application of culling algorithms in graphics software and games is widely used to optimise the performance of complex environments. Three such algorithms are outlined and explained in the literature review, which explains that Z Culling benefits less in dense environments, and that Frustum Culling struggles with complex environments if unoptimised. Much literature is outlined which employs the use of a singular optimised culling algorithm, whereas less literature taking a combined approach is believed to exist.

The research question this work set out to answer was; ”To What Extent Can the Performance of a Real Time Graphics Environment be Improved Through Combined Culling Techniques?” The combined culling algorithm approach used in this paper provides an answer to the research question, suggesting that environments can be optimised to a great extent with a combined culling algorithm approach. Ultimately, the key points which this experiment highlights are:

- Combined culling algorithms optimise non-dyanmic environments to the greatest extent
- Utilising unoptimised culling algorithms in a dynamic improves the performance to the least extent
- Frustum culling improves performance to greater extents than back face culling and Z Culling.
- Z Culling improves performance to a significantly lesser extent than the other culling algorithms
- The change in decrease in polygons lessens with more culling algorithms applied, implying a limit to how far a combined approach can go.

This research could have a significant impact in optimising graphics software, simulations, and games programs. While this experiment used unoptimised algorithms, a combined approach with fully optimised algorithms instead, could significantly better results, and improve the performance of graphics simulations to a much larger extent. This could allow medical simulations to run at higher frame rates, without compromising quality, higher end games to run on lower end hardware well, and for more complex architectural designs to be made without lowering the frame rate of the software, improving the quality and ease of use.

D. Recommendations

To answer this research question, an approach was taken which was unnecessarily complex. For others attempting to further research into this question, a simpler approach would be ideal, which does not contain as large a number of variables and factors. This prevents the issue of having exponentially larger configurations which this experiment has. This could take the form of measuring the correlation between frame rate and average polygon count, for which analysis methods such as a spearman's correlation test would be appropriate. This would help to show a more accurate correlation between the number of culling algorithms applied, and the average frames per second.

Another recommendation for those aiming to answer the same research question, would be to apply different culling algorithms. For example, occlusion culling is a common algorithm which was excluded from this experiment due to time and scope constraints.

E. Future Work

As mentioned in section VIII, the culling algorithms used in this experiment are unoptimised. Implementing optimisations to each of the culling algorithms was beyond scope for this experiment, therefore a good avenue for future work would be to apply multiple optimised culling algorithms in varying environments, to test to what extent they could improve the frame rate.

This experiment also made no attempt to vary the initial model count, with the only changes coming from the culling algorithms. The correlation between environments with lower and higher model counts could be further researched, to determine if specific culling algorithms perform better or worse in such environments.

X. References

References

[1] A. R. Forrest, "Future trends in computer graphics: How much is enough?," *Journal of Computer Science and Technology*, vol. 18, no. 5, pp. 531–537, 2003.

[2] E. Vasiou, K. Shkurko, I. Mallett, E. Brunvand, and C. Yuksel, "A detailed study of ray tracing performance: render time and energy cost," *The Visual Computer*, vol. 34, pp. 875–885, 2018.

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

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

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

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

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

[8] 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.

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

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

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

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] E. De Lucas, P. Marcuello, J.-M. Parcerisa, and A. Gonzalez, "Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 473–485, 2018.

[29] D. Corbalán-Navarro, J. L. Aragón, M. Anglada, E. de Lucas, J.-M. Parcerisa, and A. González, "Omega-test: A predictive early-z culling to improve the graphics pipeline energy-efficiency," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 12, pp. 4375–4388, 2022.

[30] 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].

[31] 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.
[32] K. et al., “Manifesto for agile software development.” Available: https://agilemanifesto.org/. Accessed 09/12/2024. [Online].
[33] 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].
[34] T. R. Foundation, “The r project for statistical computing.” Available: https://www.r-project.org/. Accessed 09/12/2024. [Online].
[35] D. Blythe, “Rise of the graphics processor,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–778, 2008.
[36] 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].
[37] “Glfw - an open gl library.” https://www.glfw.org/. Accessed 09/12/2024. [Online].
[38] “blender.org.” https://www.blender.org/. Accessed 28/03/2025. [Online].
[39] “Glad opengl library.” https://glad.dav1d.de/. Accessed 11/12/2024. [Online].
[40] “Glm mathematics library.” https://github.com/g-truc/glm. Accessed 11/12/2024. [Online].
[41] “Assimp asset importer.” https://github.com/assimp/assimp. Accessed 11/12/2024. [Online].
[42] “Appendix iii: Society for simulation in healthcare ‘healthcare simulationist code of ethics’ (1).” https://uclpartners.com/lsn-faculty/appendix-iii-society-for-simulation-in-healthcare-healthcare-simulationist-code-of-ethics-1/. Accessed 22/01/2025. [Online].
[43] “Code of conduct on artificial intelligence in military systems.” https://www.hdcentre.org/wp-content/uploads/2021/08/AI-Code-of-Conduct.pdf. Accessed 22/01/2025. [Online].
[44] “https://dplyr.tidyverse.org/index.html.” https://dplyr.tidyverse.org/index.html. Accessed 02/04/2025. [Online].
[45] “https://ggplot2.tidyverse.org/.” https://ggplot2.tidyverse.org/. Accessed 02/04/2025. [Online].
[46] D. B. Rubin, *Statistical analysis with missing data.* Wiley, 1987.

XI. Addendum

A. Statistics Addendum

Figures 13, 14, 15 and 16 show the ANOVA and linear regression analyses which were planned and conducted for this experiment. Figures 17 and 18 show the post hoc analyses done on the data after it was collected, proving that a valid sample size was taken.

The code for the Data analysis was coded using R, and the entire script can be found in subsection XII-A1 in the appendix.

The first part of the code is responsible for assigning variables to all of the imported CSV data sets. It was repeated for each data set imported:

```
env0NoCulling <- read.csv
 ("Env_0_NoCulling_Averages.csv") %>%
mutate(
 Env = "DENSE",
 BFC = 0,
 FVC = 0,
 ZC = 0 )
```

read.csv will read a csv file found in the same directory, and is a standard R function. The ”%>%” operator is included with the ”dplyr” [44] library, which adds functions for many common R related problems. The pipe (”%>%”) operator was used to streamline the process of running functions or commands on a data set, and allows the dataset declared before the operator to be manipulated by the command after the operator. The command used was the ”mutate()” command, which adds new variables to existing data sets. This was used to add the chosen environment as a string, to make reading the data easier. It was also used to add the Back Face Culling, Frustum View Culling, and Z Culling binaries to the data frame, with each of these values being hard coded as a 1 or 0 depending on the data set being added.

The next lines of code were responsible for binding all the individual data sets into one large data frame. This made it easier to run a statistical significance test on.

```
dataFrame <- rbind(
 env0NoCulling, env1NoCulling,
 env2NoCulling, env0Frustum,
 env1Frustum, env2Frustum,
 env0Back, env1Back,
 env2Back, env0Z,
 env1Z,env2Z,
 env0FrustumBack, env1FrustumBack,
 env2FrustumBack, env0FrustumZ,
 env1FrustumZ, env2FrustumZ,
 env0BackZ, env1BackZ,
 env2BackZ, env0All,
 env1All, env2All)
```

the ”rbind()” function is a simple function built into R, which binds all the passed in data sets.

The next two lines run the statistical significance test on the newly bound data frame, and output a summary of the stats test:

```
model <- lm(
 avg..fps ~ avg..polys +
 avg..models +
 Env +
 BFC +
 FVC +
 ZC, dataFrame)
summary(model)
```

This assigns the linear regression test to the variable ”model”, with the summary() command then running a summary on the statistics test. It outputs the following data, which are discussed in section VIII:

The next lines of code are for graphing the results to answer each of the hypotheses. The following code was used for hypotheses 1 and 2.

```
dataFrame <- dataFrame %>%
mutate(activeAlgorithms = FVC + BFC + ZC)
ggplot(
dataFrame,
aes(x = factor(activeAlgorithms),
y = avg..models)) +
 stat_summary(
 fun = mean,
 geom = "point",
 size = 3) +
 stat_summary(
 fun = mean, geom = "line",
 group = 1) +
 labs(
 x = "Number of active
 culling algorithms",
 y = "average rendered models",
 title = "Model count against
 number of active culling algorithms")+
 theme_bw()
```

The code first mutates the data frame to create a value which tracks the total number of active culling algorithms. It works by taking the binary values of existing culling algorithm variables, and adds them. It then plots the data frame using the ”ggplot()” command, which is from the ”ggplot2” [45] library. The graph is set to be a ”point” graph and a ”line” graph combined, to show average values specifically, and the correlation of the graph. The code for graphing the results to answer hypotheses 3 and 4 are similar to this code, and can be found under the ”Hypothesis3” and ”Hypothesis4” functions in subsection XII-A1 in the appendix.

The next parts of the data analysis code were responsible for finding the EScore value of each culling algorithm. The first lines of code find the mean values for the baseline frame rates and polygon counts.

```
avgBaselineFPS <- mean(
 defaultValues$avg..fps)
avgBaselinePolys <- mean(
 defaultValues$avg..polys)
```

This code was followed by lines of code to get the average frame rates and polygon counts for each culling algorithm combination.

```
avgBFCFPS <- mean(
dataFrame$avg..fps[
dataFrame$BFC == 1 &
dataFrame$FVC == 0 &
dataFrame$ZC == 0])
```

The example code here shows the code for getting the average frame rate for back face culling. The three binary values indicate whether or not the culling algorithm was active, and were adjusted accordingly for each culling algorithm combination. Similar code was used for finding the mean polygon count.

Finally, the EScore was calculated for each culling algorithm combination using the previously defined equations, and the variables gathered in the code.

```
BFCEScore <- ((
avgBFCFPS / avgBaselineFPS) * 100) /
((avgBFCPolys / avgBaselinePolys) * 100)
```

In the case of the Back Face Culling algorithm shown in the code, the average frames per second is divided by the average baseline frames per second, and multiplied by 100. The whole thing is then divided by the average polygons divided by the average baseline polygons, again multiplied by 100. This produced an EScore of 2.29 for back face culling, in comparison to the baseline EScore of 1.0.

B. Artefact Testing Addendum

The following code is for each of the six functions which underwent unit tests for this experiment. The independent variables (culling algorithms and environments) were tested due to their relation with the results. The standard visual studio C++ framework was used for unit tests, because it was the IDE used to develop the artefact as well.

```
TEST_METHOD(UnitTest_ZCulling)
{
 try
 {
  glm::vec4 dummyCamPos(
        0.0f, 0.0f, 1000.0f, 1.0f);
  int retFlag = 0;
  unsigned long int dummyModelsCulled = 1;

  RunZCulling(
        dummyCamPos, retFlag,
        dummyModelsCulled);
 }
 catch (const std::exception&)
 {
  Assert::Fail();
 }
}

TEST_METHOD(UnitTest_BackfaceCulling)
{
 try
 {
  RunBackFaceCulling(testWindow);
 }
 catch (const std::exception&)
 {
  Assert::Fail();
 }
}

TEST_METHOD(UnitTest_FrustumCulling)
{
 try
 {
  std::cout <<
        "Running Unit Test" <<
        std::endl;

  DummyModel testModel;
  if (!testModel.isloaded())
   Assert::Fail();
  else
  {
   std::cout <<
           "test model loaded" <<
           testModel.isloaded() <<
           std::endl;
  }

  BoundingBoxObjectClass
        testBoundingBox(testModel);

  Camera testCamera(glm::vec3(
        0.0f,
        0.0f,
        3.0f));
  Frustum testFrustum =
        CreateCameraBounds(testCamera, 1.0f,
        glm::radians(45.0f), 0.1f, 1000000.0f);

  Shader testShader(
        "shader.vs", "shader.fs");

  unsigned int testDisplay = 0;
  unsigned int testTotal = 0;

  RunFrustumCulling(testBoundingBox,
        testFrustum, testShader,
        testDisplay, testTotal);

  std::cout << "Finished Unit Test" << std::endl;
 }
 catch (const std::exception& ex)
 {
  std::cerr << ex.what() << std::endl;
  Assert::Fail();
 }
}

TEST_METHOD(UnitTest_DenseEnv)
{
 try
 {
  DummyModel testModel;
  BoundingBoxObjectClass
        testBoundingBox(testModel);

  Camera testCamera(
        glm::vec3(0.0f, 0.0f, 3.0f));
  Frustum testFrustum = CreateCameraBounds(
        testCamera, 1.0f,
        glm::radians(45.0f), 0.1f,
        1000000.0f);

  Shader testShader(
        "shader.vs", "shader.fs");
  unsigned int testDisplay;
  unsigned int testTotal;
  glm::mat4 testView = 1.0f;

  DrawDenseEnvironment(
        testBoundingBox,
        testFrustum, testShader,
        testDisplay, testTotal,
        testView, testModel);
 }
 catch (const std::exception& ex)
 {
  std::cerr << ex.what() << std::endl;
  Assert::Fail();
 }
}

TEST_METHOD(UnitTest_SparseEnv)
{
 try
 {
  DummyModel testModel;
  BoundingBoxObjectClass
        testBoundingBox(testModel);

  Camera testCamera(
        glm::vec3(0.0f, 0.0f, 3.0f));
  Frustum testFrustum = CreateCameraBounds(
        testCamera, 1.0f,
        glm::radians(45.0f), 0.1f,
        1000000.0f);

  Shader testShader(
        "shader.vs", "shader.fs");
  unsigned int testDisplay;
  unsigned int testTotal;
  glm::mat4 testView = 1.0f;

  DrawSparseEnvironment(
        testBoundingBox,
        testFrustum, testShader,
        testDisplay, testTotal,
        testView, testModel);
 }
 catch (const std::exception& ex)
 {
  std::cerr << ex.what() << std::endl;
  Assert::Fail();
 }
}

TEST_METHOD(UnitTest_DynamicEnv)
{
 try
 {
  DummyModel testModel;
  BoundingBoxObjectClass
        testBoundingBox(testModel);
  Camera testCamera(
        glm::vec3(0.0f, 0.0f, 3.0f));
  Frustum testFrustum = CreateCameraBounds(
        testCamera, 1.0f,
        glm::radians(45.0f), 0.1f,
        1000000.0f);

  Shader testShader("shader.vs", "shader.fs");
  unsigned int testDisplay;
  unsigned int testTotal;
        float testCurrentFrame = 1.0f;
  glm::mat4 testView = 1.0f;

  DrawDynamicEnvironment(
        testBoundingBox,
        testFrustum, testShader,
        testDisplay, testTotal,
        testView, testModel);
 }
 catch (const std::exception& ex)
 {
  std::cerr << ex.what() << std::endl;
  Assert::Fail();
 }
}
```

C. Critical Addendum

The development process of this dissertation was a challenging one, with new gaps in knowledge and challenges becoming prevalent every week. My relative inexperience in utilising a graphics API, and applying theoretical work to it lead to some inefficiencies in the artefact. I also lacked knowledge in more general areas such as statistics, and data analysis, leading to delays in the dissertation write up because of requiring prior research into areas such as hypothesis testing, stats testing, and data management. Mentally, the project was frustrating to work on at times, which lead to poor code quality and practices. Furthermore, it lead to a lack of engagement and enthusiasm for working on the project, resulting in an inconsistent workflow, and a ”bare minimum” approach to work. Personally, my interpersonal skills relating to the development of this artefact were poor, and my engagement with my module supervisor, and the university staff kept to a minimum. Finally, the artefact has some poorly written code, which could bring into question the ”quality” of the artefact.

1) Practical Graphics Knowledge: I decided to use OpenGL and C++ to create this artefact, because the ability to use C++ with a graphics API is a highly sought after skill in the industry for graphics programmers. Doing so has given me excellent experience in this area. The problem relating to this experiment however, was that I had relatively little experience in this area beforehand. Personally, I would like to delve deeper into graphics programming, and potentially start a career in it, and this lack of applied experience is a significant obstacle.

The experiment artefact was scaled down from its original scope, particularly with one specific culling algorithm. Initially, I wanted to implement and apply an occlusion culling algorithm, which was instead replaced by a much simpler Z-Culling algorithm. Occlusion culling is a far more popular culling algorithm to implement in graphics settings, however I was unable to implement it because of my inability to translate my theoretical knowledge of the depth buffer and view matrices into code. This stemmed from a lack of understanding of how the graphics render pipeline practically works.

To improve upon this skill, I will implement an occlusion culling algorithm within a basic 3D graphics environment, without any other external factors. To achieve this, I will further research the OpenGL documentation, specifically relating to how to utilise the depth buffer and graphics render pipeline more effectively. The goal will be to implement the culling algorithm successfully. Attempting this will build upon my existing knowledge, and reinforce my ability to apply theory in practice, a skill which many graphics programming jobs look for. This task will be carried out between the start of June, and the end of July. This gives me time to revise theory work, the OpenGL documentation, and make use of aid from university lecturers if necessary.

2) Experimental Knowledge Gaps - Hypothesis Testing, Stats Collection etc.: A major challenge for me during this experiment was the lack of understanding of statistics and data. There is a gap in my knowledge relating to statistical relevance tests, hypothesis tests, and scientific data collection, leading to my tests and hypotheses not being as strong as they could have been. Skills pertaining to this are relevant in the games industry, because new methods must first be experimented upon, which require a strong understanding of data analysis to yield sufficient results to be implemented.

The data gathered in this experiment, while done to a satisfactory standard, could have been statistically analysed in greater detail, revealing further insights and answers to the research question. Furthermore, the hypotheses, again while acceptable, were not tested. If care had not been taken, this could have lead to the irrelevant hypotheses, or hypotheses which did not provide any further insight into the research question. This was a result of my lack of knowledge pertaining to statistical significance testing, data analysis, and hypothesis testing.

To improve upon this skill, I will read ”Statistical Analysis with Missing Data” [46] by Rubin et al. It focuses on topics such as Data analysis, analysis methods, algorithms and more content which is relevant to all aspects of data analysis. This book is available online, and free to access from the Falmouth University library, making it easily accessible for me. While reading this, I will also proactively take and summarise notes to help cement my understanding in the subject. This will allow me to be more confident in proposing hypotheses, and analysing data going forward. I will aim to read this book over the summer holidays, between May-August.

3) Bug Frustration: As is common for coding projects, unexpected bugs appearing were a frequent occurrence during the development lifecycle of the artefact. When under a supposed timescale of roughly three weeks to develop the artefact, these bugs quickly become frustrating to deal with. This frustration then leads to poorly made decisions, and improper practices. Dealing with this frustrations in a professional manner, and keeping a clear and logical thought process when potential fixes fails is a crucial skill for my ideal career path in the games industry, because industry standards must be upheld regardless of potential frustration, otherwise team work may be negatively affected.

Some of the code in the artefact, and commit messages in the repository is poorly maintained, or not appropriately named. The poor code maintainability lead to wasted time having to re-learn, or figure out what work I had done previously. The inappropriately named commit messages resulted in me forgetting where I had stopped working previously, therefore losing time having to figure out what steps I would take next, and what I would be working on for that day. These two consequences were the result of the emotional frustration felt, often when failing to fix a bug. During this frustration, it is easy to forget proper practices, and rush a solution to completion without proper documentation.

To improve on this area of development, I will be more disciplined in my approach to fixing bugs, while also being more self aware of my current emotions. While working, I will aim to adequately document code by each function at least. Furthermore, If I am attempting to fix a bug, and am unable to, I will take a break, and work on something else rather than repeating attempted fixes and building upon frustration. This will allow me to create a better work flow for myself, and in group working scenarios, will not damage team harmony as poorly. This will be a continuous practice that will be worked on indefinitely throughout my career.

4) Project Engagement: Throughout the early stages of the projects development, my engagement with the content was minimal. I believe this happened because I did not incorporate Agile development principles early in development, which lead to a form of procrastination & unwillingness to start work, due to the seemingly enormous scope of the project. This is further backed by the fact that later in the project, I started incorporating Agile development principles, and broke down the task into smaller, more manageable chunks. From here, my development on the project became significantly more consistent, and I was able to work on the project for at least 20 hours per week.

The early procrastination resulted in a later rush to catch up, which lead to a hurriedly proposed research question. This in turn, had a knock on effect with my entire experiment, which was then trying to answer a vague question. This happened because I did not break down the tasks required in the early stages of the development cycle. Therefore, I was significantly less engaged with the project, and left deciding on a research question until far too late.

To improve on this skill and area, I will aim to implement Agile and Scrum practices from the very beginning of a projects development lifecycle. This will include breaking down the main goal into smaller manageable tasks. I will also aim to create design documents for the artefacts I create. This is relevant to me, because it will prevent procrastination due to the project scope, and help me become more productive overall. This will be an ongoing practice

5) Lack of engagement with supervisor and staff: I did not engage with my dissertation supervisor, the university staff, or my peers very much throughout the development process. On one hand, this made the act of completing such a project more personally satisfying, because it feels as though it was completed more of my own merit. The problem with this approach however, was that I made many significant errors while designing, and writing up the experiment. This is a very important skill to work on, because many junior and graduate positions in industries often have a senior or mentor to help if necessary. In this roles, you are expected to ask for help when necessary, and it is often better to ask for help and get the job correct the first time, instead of getting it wrong.

The lack of seeking help and feedback, resulted in a poorly designed, and complex experiment which involved two independent variables. The research question could have realistically been answered with an experiment including only one independent variable. Furthermore, this complex design resulted in complications in the write up, leading to the initial grade for the project proposal scoring quite lowly.

To improve on this skill and area, I will aim to ask for help from those more senior more frequently when struggling. I will apply this to my current team project. If I come across an issue, I will first aim to spend at least 30 minutes fixing the issue independently. If I am successful, I will ask my teammates, or any lecturers for assistance. This will take place immediately, and last until the end of May, the team projects conclusion.

6) Code Quality: The artefact created in this project makes use of many scripts, each with complex features. It was made using C++, a language in which I have relatively little experience when compared to other languages I am proficient in. Some sections of the project are robust and coded well, such as the many header files, classes, and object oriented approach taken. However, other sections are poorly designed, and duplicate code can be found. This is a relevant problem for me to tackle, because in the industry, code is expected to be to a professional, and robust level at all times. Getting a job in the industry very much relies on my ability to write high quality, robust code.

The poor code quality in some areas, resulted in a poorly optimised culling algorithm and environment combination being implemented. This negatively impacted the results of my experiment, resulting in the research question not being answered to as thorough an extent as it could have. This happened because of my inability to implement a high quality and efficient solution in time, which in turn happened because of my lack of C++ knowledge.

To remedy this issue, I will further familiarise myself with C++, and aim to code a new graphics project using C++ to help further my knowledge. I will first learn how to make use of memory management in C++ - a key feature. Then, I will aim to create a project using C++ and memory allocation to cement my knowledge. To measure progress, I will make of the previously mentioned Agile practice to break down the project scope into smaller tasks, which will be tracked to measure progress. This Task will be performed between May-July 2025, giving me 1 month to gain a better knowledge base with C++, and another month to create the project.

7) Conclusion: Ultimately, I have learned a great deal from this project, and my experience and knowledge in industry and potential career relevant practices has greatly increased. In hindsight, this project was vastly overcomplicated, and could have been simplified while still answering a valid research question. This was caused by a combination of a lack of knowledge required, a lack of proper Agile development methodology from the very beginning of the project, and a lack of interpersonal communication between myself, and lecturers and staff who could have assisted me.

From this experience, I have learned the importance of knowing my own limitations, and how it can negatively affect a personally ambitious project. I have also learned first hand, the effectiveness of Agile and Scrum development methodology as a means of motivation and engagement, and therefore its importance in proper application for software development. Furthermore, I have also learned the importance of being able to ask for help in difficult situations, because some of the best learning experiences for me throughout this project were when I asked for help. To improve upon my flaws, I will apply agile practices in the early ideation stage of a project, further my knowledge of data analysis and statistics, further my knowledge of graphics programming, and C++ as a language, communicate with others more effectively, asking for help where necessary, and finally, I will aim to create more graphics related projects in C++, to apply the previously listed improvements practically.

XII. Appendix

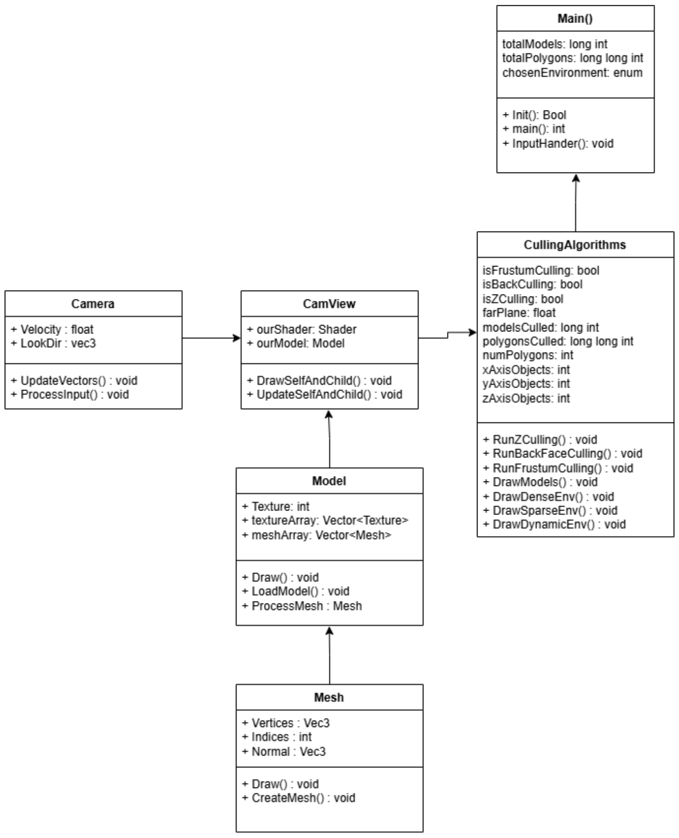
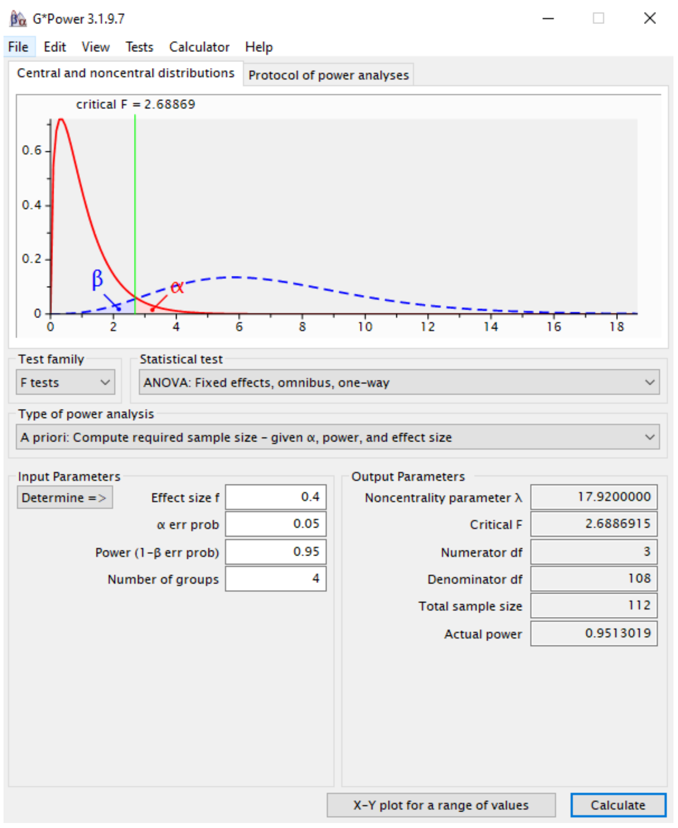


Fig. 12. Class diagram for the artefact


G*Power 3.1.9.7
File Edit View Tests Calculator Help
Central and noncentral distributions
Protocol of power analyses
critical F = 2.68869
β
α
X-Y plot for a range of values
Calculate

Test family: F tests

Statistical test: ANOVA: Fixed effects, omnibus, one-way

Type of power analysis: A priori: Compute required sample size – given α, power, and effect size

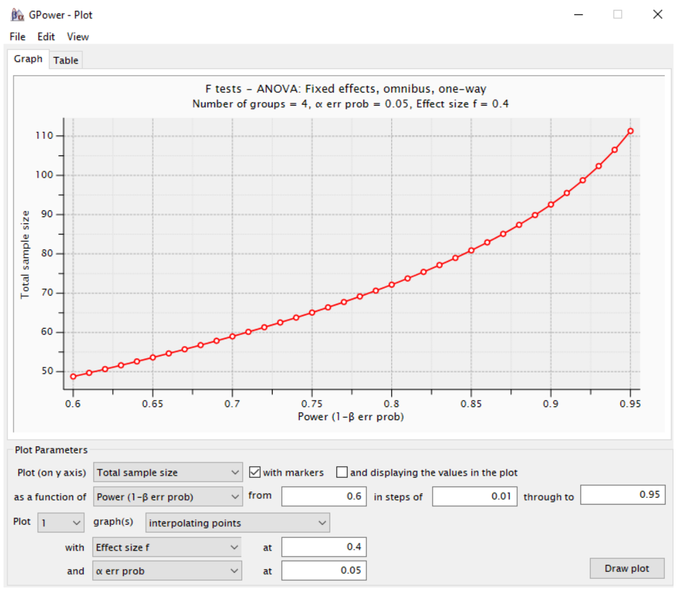
Input Parameters

Effect size f	0.4
α err prob	0.05
Power (1-β err prob)	0.95
Number of groups	4

Output Parameters

Noncentrality parameter λ	17.9200000
Critical F	2.6886915
Numerator df	3
Denominator df	108
Total sample size	112
Actual power	0.9513019

Fig. 13. A priori ANOVA test stats



Plot Parameters

Plot (on y axis): Total sample size ☑ with markers ☐ and displaying the values in the plot

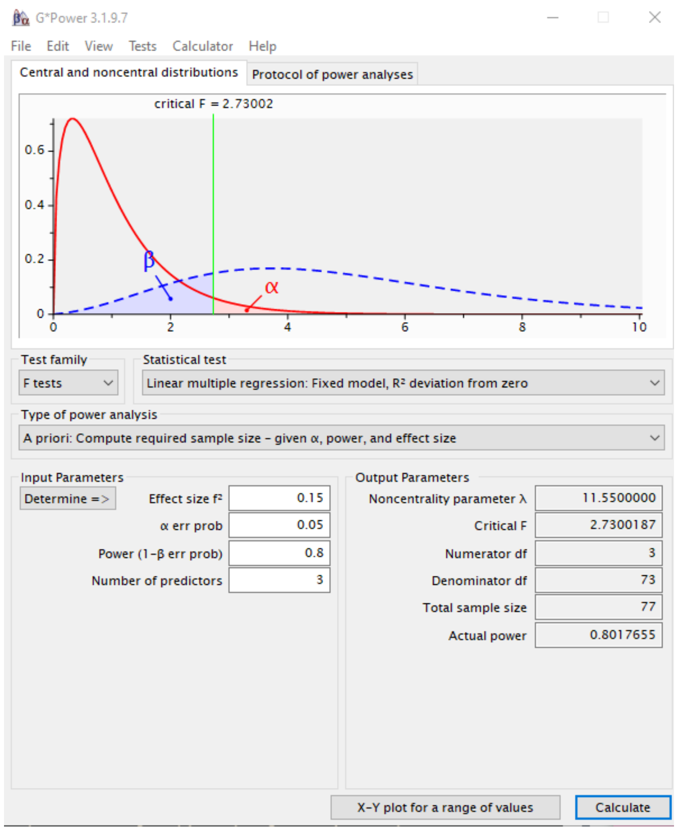
as a function of: Power (1-β err prob) from 0.6 in steps of 0.01 through to 0.95

Plot 1 graph(s) interpolating points

with Effect size f at 0.4

and α err prob at 0.05

Fig. 14. A priori ANOVA type 2 error probability


G*Power 3.1.9.7
File Edit View Tests Calculator Help
Central and noncentral distributions
Protocol of power analyses
critical F = 2.73002
β
α
X–Y plot for a range of values
Calculate
Determine =>

Test family: F tests

Statistical test: Linear multiple regression: Fixed model, R² deviation from zero

Type of power analysis: A priori: Compute required sample size – given α, power, and effect size

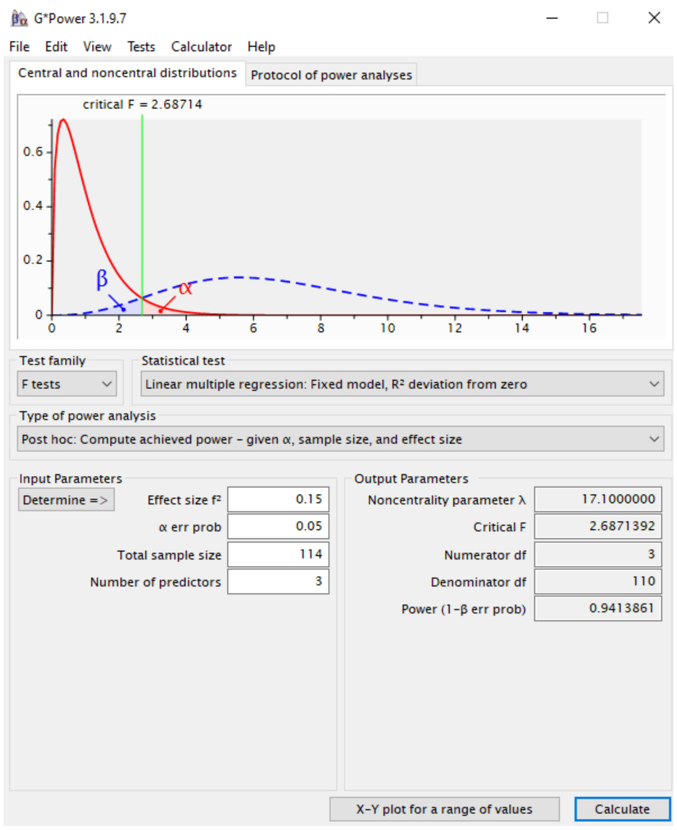
Input Parameters

Effect size f²	0.15
α err prob	0.05
Power (1–β err prob)	0.8
Number of predictors	3

Output Parameters

Noncentrality parameter λ	11.5500000
Critical F	2.7300187
Numerator df	3
Denominator df	73
Total sample size	77
Actual power	0.8017655

Fig. 15. A Priori Lienar Regression G*Power test


G*Power 3.1.9.7
File Edit View Tests Calculator Help
Central and noncentral distributions
Protocol of power analyses
critical F = 2.68714
β
α
X–Y plot for a range of values
Calculate
Determine =>

Test family: F tests

Statistical test: Linear multiple regression: Fixed model, R² deviation from zero

Type of power analysis: Post hoc: Compute achieved power – given α, sample size, and effect size

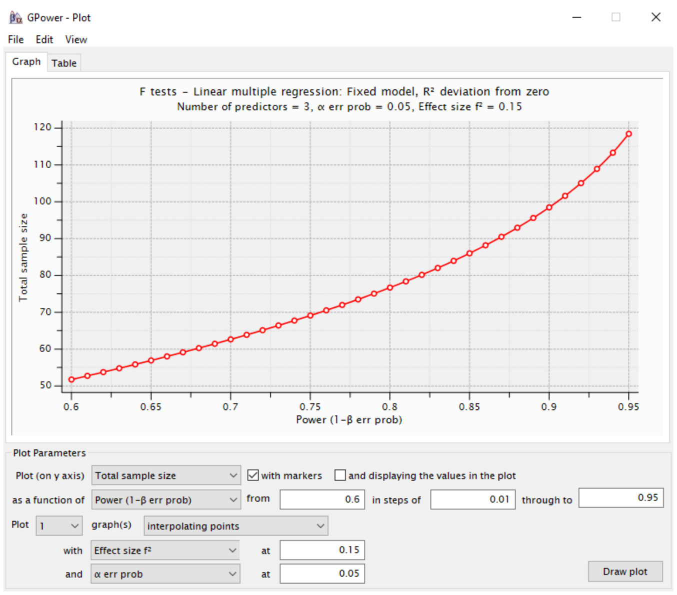
Input Parameters

Effect size f²	0.15
α err prob	0.05
Total sample size	114
Number of predictors	3

Output Parameters

Noncentrality parameter λ	17.1000000
Critical F	2.6871392
Numerator df	3
Denominator df	110
Power (1–β err prob)	0.9413861

Fig. 17. Post hoc analysis


GPower - Plot
File Edit View
Graph Table
F tests – Linear multiple regression: Fixed model, R² deviation from zero
Number of predictors = 3, α err prob = 0.05, Effect size f² = 0.15
Total sample size
Power (1–β err prob)
Draw plot

Plot Parameters

Plot (on y axis): Total sample size; with markers [✓]; and displaying the values in the plot [ ]

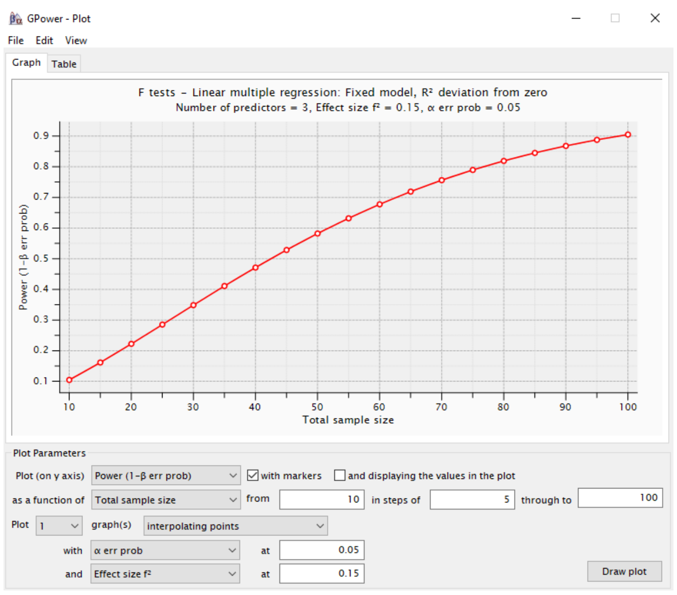
as a function of: Power (1–β err prob) from 0.6 in steps of 0.01 through to 0.95

Plot 1 graph(s) interpolating points

with Effect size f² at 0.15

and α err prob at 0.05

Fig. 16. A Priori Linear Regression type 2 error probability


GPower - Plot
File Edit View
Graph Table
F tests – Linear multiple regression: Fixed model, R² deviation from zero
Number of predictors = 3, Effect size f² = 0.15, α err prob = 0.05
Power (1–β err prob)
Total sample size
Draw plot

Plot Parameters

Plot (on y axis): Power (1–β err prob); with markers [✓]; and displaying the values in the plot [ ]

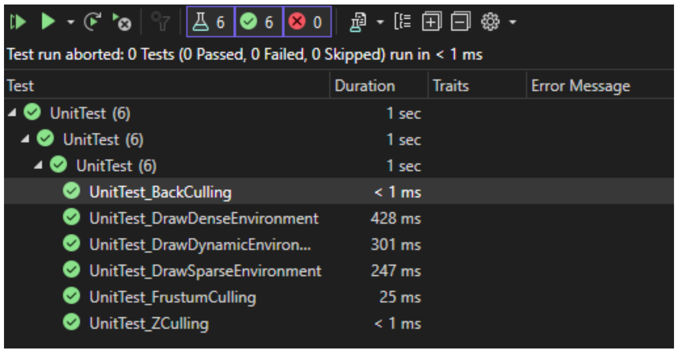
as a function of: Total sample size from 10 in steps of 5 through to 100

Plot 1 graph(s) interpolating points

with α err prob at 0.05

and Effect size f² at 0.15

Fig. 18. Post hoc graphed



6 6 0

Test run aborted: 0 Tests (0 Passed, 0 Failed, 0 Skipped) run in < 1 ms

```
Test                                 Duration   Traits   Error Message
UnitTest (6)                            1 sec
  UnitTest (6)                          1 sec
    UnitTest (6)                        1 sec
      UnitTest_BackCulling             < 1 ms
      UnitTest_DrawDenseEnvironment    428 ms
      UnitTest_DrawDynamicEnviron...   301 ms
      UnitTest_DrawSparseEnvironment   247 ms
      UnitTest_FrustumCulling           25 ms
      UnitTest_ZCulling                < 1 ms
```

Fig. 19. Validation/Unit testing

Active Culling Algorithms	avg.fps	avg.model	avg.poly
0	14.56	15625	1.92e+08
1	18.91	11690	1.11e+08
2	24.07	8166	6.09e+07
3	33.93	4848	2.97e+07

Fig. 20. value averages against number of culling algorithms active

A. Code Excerpts

```
library ( dplyr )
library ( ggplot2 )
library ( scales )

#Code for the data is a mix of basic RStudio code, and
      libraries . The library code is explained below.

# "%>%" is the pipe operator. Allows you to do stuff to
      the variable assigned on that line , makes code
      neater
# https :// magrittr . tidyverse .org/ reference /pipe . html

# "mutate()" is a dplyr function , that adds new
      variables to existing data
# https :// dplyr . tidyverse .org/

#plotting dataframes: https :// www.geeksforgeeks.org/ how
      −to−plot−all−the−columns−of−a−dataframe−in−r/
#box plots : https :// www.sthda.com/english/ wiki / ggplot2 −
      box−plot−quick−start−guide−r−software−and−data−
      visualization
#vectors : https :// www.rdocumentation.org/packages/base/
      versions / 3.6.2 / topics / c

#This function reads all the files and assigns them to
      variables
Func_ReadFiles <− function()
{
  env0NoCulling <− read.csv("Env_0_NoCulling_Averages
      .csv") %>% #reads the csv file
    mutate(Env = "DENSE", BFC = 0, FVC = 0, ZC = 0)
        #adds new headings − binaries idea for culling
        algorithms given by Michael Scott (thankyou!)
  env1NoCulling <− read.csv("Env_1_NoCulling_Averages
      .csv") %>%
    mutate(Env = "SPARSE", BFC = 0, FVC = 0, ZC = 0 )
  env2NoCulling <− read.csv("Env_2_NoCulling_Averages
      .csv") %>%
    mutate(Env = "DYNAMIC", BFC = 0, FVC = 0, ZC =
        0)

  env0Frustum <− read.csv("Env_0_FrustumCulling_
      Averages.csv") %>%
    mutate(Env = "DENSE", BFC = 0, FVC = 1, ZC = 0)
  env1Frustum <− read.csv("Env_1_FrustumCulling_
      Averages.csv") %>%
    mutate(Env = "SPARSE", BFC = 0, FVC = 1, ZC = 0)
  env2Frustum <− read.csv("Env_2_FrustumCulling_
      Averages.csv") %>%
    mutate(Env = "DYNAMIC", BFC = 0, FVC = 1, ZC =
        0)

  env0Back <− read.csv("Env_0_BackfaceCulling_
      Averages.csv") %>%
    mutate(Env = "DENSE", BFC = 1, FVC = 0, ZC = 0)
env1Back <− read.csv("Env_1_BackfaceCulling_
      Averages.csv") %>%
  mutate(Env = "SPARSE", BFC = 1, FVC = 0, ZC = 0)
env2Back <− read.csv("Env_2_BackfaceCulling_
      Averages.csv") %>%
  mutate(Env = "DYNAMIC", BFC = 1, FVC = 0, ZC =
      0)

env0Z <− read.csv("Env_0_ZCulling_Averages.csv")
      %>%
  mutate(Env = "DENSE", BFC = 0, FVC = 0, ZC = 1)
env1Z <− read.csv("Env_1_ZCulling_Averages.csv")
      %>%
  mutate(Env = "SPARSE", BFC = 0, FVC = 0, ZC = 1)
env2Z <− read.csv("Env_2_ZCulling_Averages.csv")
      %>%
  mutate(Env = "DYNAMIC", BFC = 0, FVC = 0, ZC =
      1)

env0FrustumBack <− read.csv("Env_0_
      BackfaceAndFrustum_Averages.csv") %>%
  mutate(Env = "DENSE", BFC = 1, FVC = 1, ZC = 0)
env1FrustumBack <− read.csv("Env_1_
      BackfaceAndFrustum_Averages.csv") %>%
  mutate(Env = "SPARSE", BFC = 1, FVC = 1, ZC = 0)
env2FrustumBack <− read.csv("Env_2_
      BackfaceAndFrustum_Averages.csv") %>%
  mutate(Env = "DYNAMIC", BFC = 1, FVC = 1, ZC =
      0)

env0FrustumZ <− read.csv("Env_0_FrustumAndZ_
      Averages.csv") %>%
  mutate(Env = "DENSE", BFC = 0, FVC = 1, ZC = 1)
env1FrustumZ <− read.csv("Env_1_FrustumAndZ_
      Averages.csv") %>%
  mutate(Env = "SPARSE", BFC = 0, FVC = 1, ZC = 1)
env2FrustumZ <− read.csv("Env_2_FrustumAndZ_
      Averages.csv") %>%
  mutate(Env = "DYNAMIC", BFC = 0, FVC = 1, ZC =
      1)

env0BackZ <− read.csv("Env_0_BackfaceAndZ_
      Averages.csv") %>%
  mutate(Env = "DENSE", BFC = 1, FVC = 0, ZC = 1)
env1BackZ <− read.csv("Env_1_BackfaceAndZ_
      Averages.csv") %>%
  mutate(Env = "SPARSE", BFC = 1, FVC = 0, ZC = 1)
env2BackZ <− read.csv("Env_2_BackfaceAndZ_
      Averages.csv") %>%
  mutate(Env = "DYNAMIC", BFC = 1, FVC = 0, ZC =
      1)

env0All <− read.csv("Env_0_AllCulling_Averages.csv")
      %>%
  mutate(Env = "DENSE", BFC = 1, FVC = 1, ZC = 1)
  env1All <- read.csv("Env_1_AllCulling_Averages.csv")
      %>%
    mutate(Env = "SPARSE", BFC = 1, FVC = 1, ZC = 1)
  env2All <- read.csv("Env_2_AllCulling_Averages.csv")
      %>%
    mutate(Env = "DYNAMIC", BFC = 1, FVC = 1, ZC =
        1)
}

#merge the data into one data frame
dataFrame <- rbind(env0NoCulling, env1NoCulling,
    env2NoCulling,
                   env0Frustum, env1Frustum,
                       env2Frustum,
                   env0Back, env1Back, env2Back,
                   env0Z, env1Z, env2Z,
                   env0FrustumBack, env1FrustumBack,
                       env2FrustumBack,
                   env0FrustumZ, env1FrustumZ,
                       env2FrustumZ,
                   env0BackZ, env1BackZ, env2BackZ,
                   env0All, env1All, env2All)

model <- lm(avg..fps ~ avg..polys + avg..models + Env +
    BFC + FVC + ZC, dataFrame) #assign stats test to a
    model
summary(model) #summarise the model

#creates the graphs and dataframes for hypotheses 1 and
    2
Func_Hypothesis1And2 <- function()
{
  dataFrame <- dataFrame %>% mutate(activeAlgorithms
      = FVC + BFC + ZC)
  cleanDf <- dataFrame[-221,] #clean data by removing
      an empty entry
  #model count for polygons
  polydf <- ggplot(dataFrame, aes(x = factor(
      activeAlgorithms), y = avg..polys)) +
    stat_summary(fun = mean, geom = "point", size = 3)
        + stat_summary(fun = mean, geom = "line",
        group = 1) +
    labs(x = "Number of active culling algorithms", y =
        "average rendered polygons", title = "Model
        count against number of active culling
        algorithms")+
    theme_bw()

  #get polygon count averages against number of culling
      algorithms
  GraphPolyAvg0 <- mean(cleanDf$avg..polys[cleanDf$
      activeAlgorithms == 0])
  GraphPolyAvg1 <- mean(cleanDf$avg..polys[cleanDf$
      activeAlgorithms == 1])
  GraphPolyAvg2 <- mean(cleanDf$avg..polys[cleanDf$
      activeAlgorithms == 2])
  GraphPolyAvg3 <- mean(cleanDf$avg..polys[cleanDf$
      activeAlgorithms == 3])

  #dataframe for model count
  modeldf <- ggplot(dataFrame, aes(x = factor(
      activeAlgorithms), y = avg..models)) +
    stat_summary(fun = mean, geom = "point", size = 3)
        + stat_summary(fun = mean, geom = "line",
        group = 1) +
    labs(x = "Number of active culling algorithms", y =
        "average rendered models", title = "Model
        count against number of active culling
        algorithms")+
    theme_bw()

  #get model count averages against number of culling
      algorithms
  GraphModelAvg0 <- mean(cleanDf$avg..models[cleanDf$
      activeAlgorithms == 0])
  GraphModelAvg1 <- mean(cleanDf$avg..models[cleanDf$
      activeAlgorithms == 1])
  GraphModelAvg2 <- mean(cleanDf$avg..models[cleanDf$
      activeAlgorithms == 2])
  GraphModelAvg3 <- mean(cleanDf$avg..models[cleanDf$
      activeAlgorithms == 3])

  fpsdf <- ggplot(dataFrame, aes(x = factor(
      activeAlgorithms), y = avg..fps)) +
    stat_summary(fun = mean, geom = "point", size = 3)
        + stat_summary(fun = mean, geom = "line",
        group = 1) +
    labs(x = "Number of active culling algorithms", y =
        "average frames per second", title = "Model
        count against number of active culling
        algorithms")+
    theme_bw()

  #get framerate count against number of culling
      algorithms
  GraphFPSAvg0 <- mean(cleanDf$avg..fps[cleanDf$
      activeAlgorithms == 0])
  GraphFPSAvg1 <- mean(cleanDf$avg..fps[cleanDf$
      activeAlgorithms == 1])
  GraphFPSAvg2 <- mean(cleanDf$avg..fps[cleanDf$
      activeAlgorithms == 2])
  GraphFPSAvg3 <- mean(cleanDf$avg..fps[cleanDf$
      activeAlgorithms == 3])
}

#gets graphs and data for hypothesis 3
Func_Hypothesis3 <- function()
{
  ggplot(dataFrame, aes(x = avg..polys, y = avg..fps,
      color = Env)) + geom_point() + geom_smooth(
      method =lm) + theme_bw() +
    labs(
      title = "Polygon Count against average FPS by
          environment",
      x = "Polygon Count",
      y = "Average FPS"
    )
}

#gets graphs and data for hypothesis 4 and 5
Func_Hypothesis4 <- function()
{
  cleanDf <- dataFrame[-221,]
  ggplot(dataFrame,
    aes(x = Env, y = avg..fps, fill = factor(BFC))) +
        geom_boxplot() + labs(
     title = "FPS Based on back face culling",
    x = "Environment",
    y = "Average FPS") +
    theme_minimal()

  ggplot(dataFrame,
    aes(x = Env, y = avg..fps, fill = factor(FVC))) +
        geom_boxplot() + labs(
     title = "FPS Based on Frustum View Culling",
    x = "Environment",
    y = "Average FPS") +
  theme_minimal()

  ggplot(dataFrame,
    aes(x = Env, y = avg..fps, fill = factor(ZC))) +
        geom_boxplot() + labs(
     title = "FPS Based on z Culling culling",
    x = "Environment",
    y = "Average FPS") +
  theme_minimal()

  #get subsets showing data values for these graphs and
        summarise to get stats
  bp1df <- subset(dataFrame, BFC == 1 & FVC == 0 &
       ZC == 0)
  summary(bp1df)

  bp2df <- subset(dataFrame, BFC == 0 & FVC == 1 &
       ZC == 0)
  summary (bp2df)

  bp3df <- subset(dataFrame, BFC == 0 & FVC == 0 &
       ZC == 1)
  summary (bp3df)
}

#gets escores
Func_GetEScore <- function()
{
   defaultValues <- subset(dataFrame, BFC == 0 & FVC
        == 0 & ZC == 0)
  df2 <- dataFrame[-221,] #cleaned version of the
        dataframe

#Baseline FPS and Polygon Counts
avgBaselineFPS <- mean(defaultValues$avg..fps)
avgBaselinePolys <- mean(defaultValues$avg..polys)

#Average FPS Counts

avgBFCFPS <- mean(dataFrame$avg..fps[dataFrame$
      BFC == 1 & dataFrame$FVC == 0 & dataFrame$ZC
       == 0]) #get mean depending on another value
avgFVCFPS <- mean(df2$avg..fps[dataFrame$BFC == 0
       & dataFrame$FVC == 1 & dataFrame$ZC == 0])
avgZCFPS <- mean(dataFrame$avg..fps[dataFrame$BFC
      == 0 & dataFrame$FVC == 0 & dataFrame$ZC ==
      1])

avgBFCFVCFPS <- mean(dataFrame$avg..fps[dataFrame
      $BFC == 1 & dataFrame$FVC == 1 & dataFrame$
      ZC == 0])
avgBFCZCFPS <- mean(dataFrame$avg..fps[dataFrame$
      BFC == 1 & dataFrame$FVC == 0 & dataFrame$ZC
       == 1])
avgFVCZCFPS <- mean(dataFrame$avg..fps[dataFrame$
      BFC == 0 & dataFrame$FVC == 1 & dataFrame$ZC
       == 1])

avgCombinedFPS <- mean(dataFrame$avg..fps[
      dataFrame$BFC == 1 & dataFrame$FVC ==1 &
      dataFrame$ZC == 1])

#Average Polygon Counts

avgBFCPolys <- mean(dataFrame$avg..polys[dataFrame$
      BFC == 1 & dataFrame$FVC == 0 & dataFrame$ZC
       == 0])
avgFVCPolys <- mean(df2$avg..polys[dataFrame$BFC
      == 0 & dataFrame$FVC == 1 & dataFrame$ZC ==
      0])
avgZCPolys <- mean(dataFrame$avg..polys[dataFrame$
      BFC == 0 & dataFrame$FVC == 0 & dataFrame$ZC
       == 1])

avgBFCFVCPolys <- mean(dataFrame$avg..polys[
      dataFrame$BFC == 1 & dataFrame$FVC == 1 &
      dataFrame$ZC == 0])
avgBFCZCPolys <- mean(dataFrame$avg..polys[
      dataFrame$BFC == 1 & dataFrame$FVC == 0 &
      dataFrame$ZC == 1])
avgFVCZCPolys <- mean(dataFrame$avg..polys[
      dataFrame$BFC == 0 & dataFrame$FVC == 1 &
      dataFrame$ZC == 1])

avgCombinedPolys <- mean(dataFrame$avg..polys[
      dataFrame$BFC == 1 & dataFrame$FVC == 1 &
      dataFrame$ZC == 1])
  #EScore Value  Calculation
  NoCullingEScore <- ((avgBaselineFPS / avgBaselineFPS
      ) * 100) / ((avgBaselinePolys  /  avgBaselinePolys)
      * 100)

  BFCEScore <- ((avgBFCFPS / avgBaselineFPS) * 100) /
       ((avgBFCPolys / avgBaselinePolys) * 100)

  FVCEScore <- ((avgFVCFPS / avgBaselineFPS) * 100) /
       ((avgFVCPolys / avgBaselinePolys) * 100)

  ZCEScore <- ((avgZCFPS / avgBaselineFPS) * 100) / ((
      avgZCPolys / avgBaselinePolys) * 100)

  BFCFVCEScore <- ((avgBFCFVCFPS / avgBaselineFPS
      ) * 100) / ((avgBFCFVCPolys / avgBaselinePolys) *
      100)

  BFCZCEScore <- ((avgBFCZCFPS / avgBaselineFPS) *
      100) / ((avgBFCZCPolys / avgBaselinePolys) * 100)

  FVCZCEScore <- ((avgFVCZCFPS / avgBaselineFPS) *
       100) / ((avgFVCZCPolys / avgBaselinePolys) * 100)

  CombinedEScore <- ((avgCombinedFPS /
      avgBaselineFPS) * 100) / ((avgCombinedPolys /
      avgBaselinePolys) * 100)

  EScoreDataFrame <- data.frame(
    name = c("E.Baseline",  "E.BFC", "E.FVC", "E.ZC", "
         E.BFCFVC", "E.BFCZC", "E.FVCZC", "E.All"),
    value  = c(NoCullingEScore, BFCEScore, FVCEScore,
         ZCEScore, BFCFVCEScore, BFCZCEScore,
         FVCZCEScore, CombinedEScore)
    )

  ggplot(EScoreDataFrame, aes(x = name, y = value)) +
    geom_bar(stat = " identity ")
}
Func_ReadFiles()
Func_Hypothesis1And2()
Func_Hypothesis3()
Func_Hypothesis4()
Func_GetEScore()

#csv  writing
write.csv(dataFrame, "D:/dataFrame.csv")
write.csv( fpsdf$data, "D:/FpsGraphDataValues.csv")
```