Real Time Graphics Engine 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. TODO:: BRIEF MENTION OF THE CONCLUSION HERE**

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 framerate. 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 framerate 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 VII. The data gathered is presented and visualised in this section, with statistical analysis tests also being conducted in this section.

Finally, section TODO:: ADD SECTION REFERENCE discusses 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

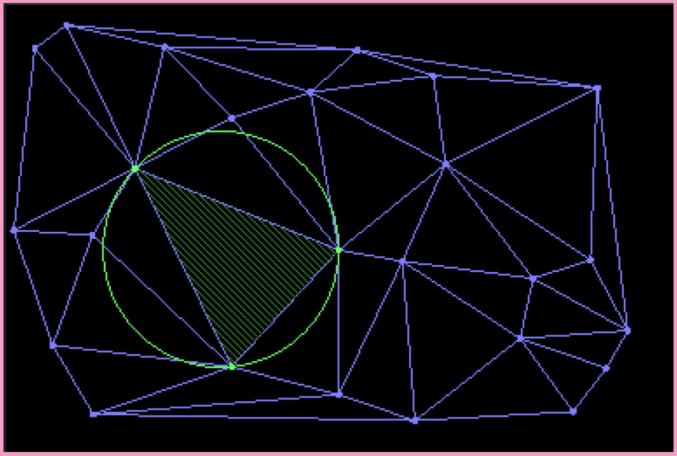


Fig. 1. An example of Delaunay triangulation used on a complex mesh. Note that the three vertices of the highlighted triangle are always on the circumference of the circle, and never inside it. [16]

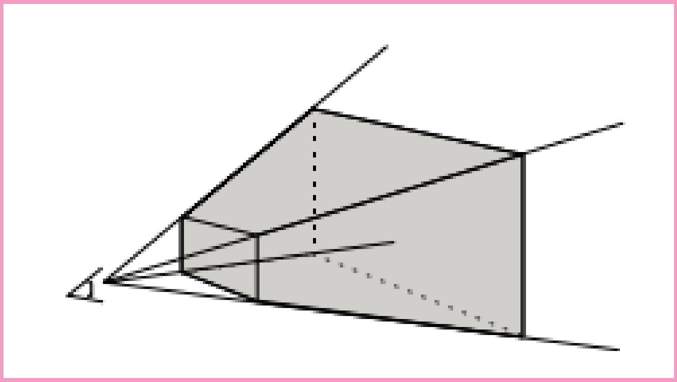


Fig. 2. The view frustum projected out from the camera view. [18].

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 alogorithms, 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 Engine 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 framerate 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 and cull the most triangles. Z culling when applied in isolation will improve performance the least.” This is working off the assumption that back-face culling should be culling approximately half of all triangles in the environment, whereas frustum view culling and Z culling depend on the viewing direction and position of the user.

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 center points.
- 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.

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, 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 recordered 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

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

users with lower spec machines than the recommended to run the game engine with an increased framerate, and lower memory and GPU usage. This would be especially useful for simulations of a medical or military nature, in which the speed of the program is paramount, because it could have potentially life threatening consequences if un-optimized.

3) reliability: The data collected from this experiment can be easily repeated, and should yield similar accurate results every time. The three culling algorithms applied to the simulations 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) replicability: This experiment, if carried out via the same method, should be very replicable. 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 replicability 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 ((TODO::REFERENCE IMAGE IN APPENDIX TO UNIT TESTS)) 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 ((TODO::REFERENCE IMAGE IN APPENDIX TO UNIT TESTS)) 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.

TODO:: DESIGN AND REFACTORING PART OF THE RUBRIC

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 framerate, 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:

- Framerate - This was used to measure the performance of the program before and after the culling algorithms are applied, with a higher framerate being more favourable than a lower framerate. 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 framerate 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 framerate 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.
- framerate: int - Dependent variable, number of frames displayed per second. This was used to measure the performance of each culling algorithm. A higher framerate 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. 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 framerate, average polygon count, and average model count to a list during runtime. Upon the programs termination, the averages of the framerate, 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 framerate, 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 framerate 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 framerate 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.

D. Data Analysis Code

The code for the Data analysis was coded using R, and the entire script can be found in subsection XI-B1 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” [42] 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 TODO:: ADD SECTION REFERENCE:

```
 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 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” [43] 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 XI-B1 in the appendix.

TODO:: Finish the R Code Analysis

VII. 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 3 and 4 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 5 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 7 & 8 returned a power of 0.941, therefore decreasing the probability of a type 2 error by 0.141.

VIII. 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. 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 [44] [45]. Conversely, a poorly optimised culling could lead to a severe drop in framerate 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.

IX. 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] “https://dplyr.tidyverse.org/index.html.” https://dplyr.tidyverse.org/index.html. Accessed 02/04/2025. [Online].

[43] “https://ggplot2.tidyverse.org/.” https://ggplot2.tidyverse.org/. Accessed 02/04/2025. [Online].

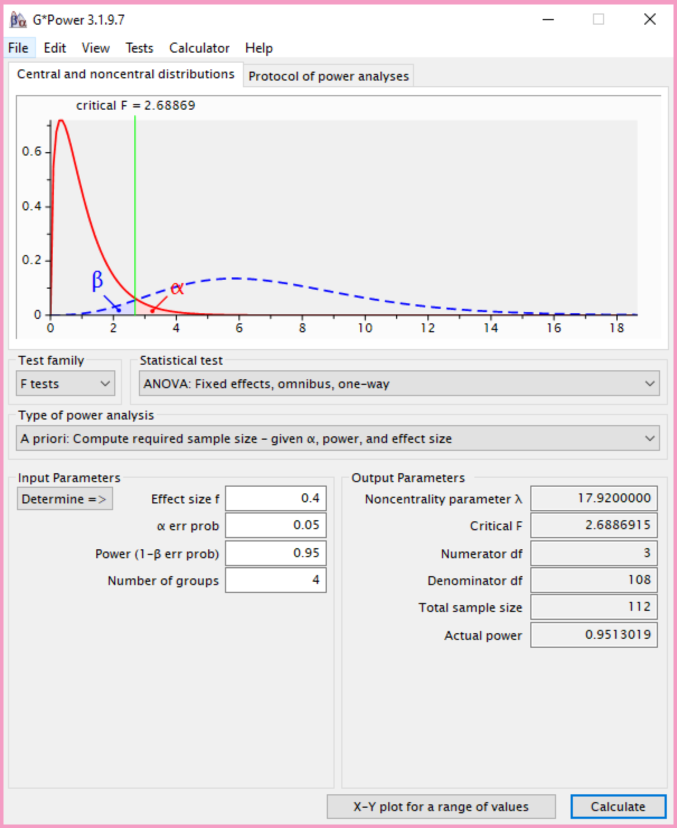
[44] “Appendix iii: Society for simulation in healthcare ‘healthcare simulationist code of ethics’ (1).” https://uclpartners.com/lsn-faculty/appendix-iii-society-for-simulation-in-healthcare-healthcare-simulationist-code-of-ethics-1/. Accessed 22/01/2025. [Online].

[45] “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].

X. Addendum

XI. Appendix

A. *Stats Tests*


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

Test family: F tests

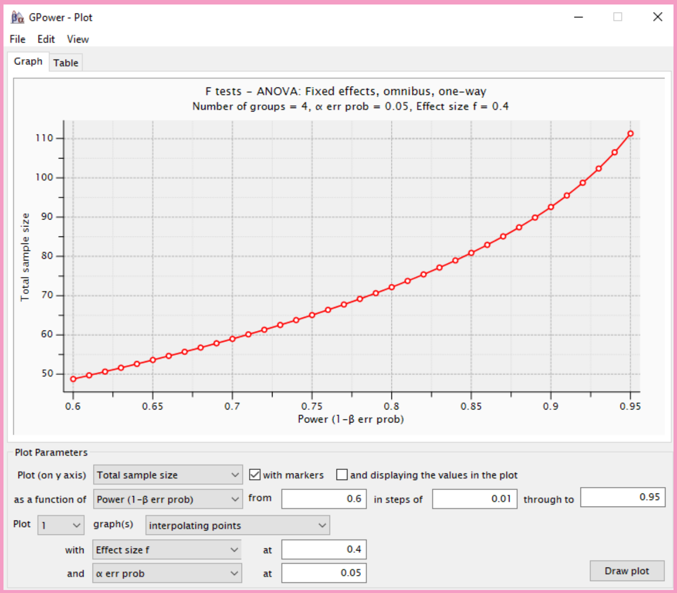
Statistical test: ANOVA: Fixed effects, omnibus, one–way

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. 3. A priori ANOVA test stats



F tests – ANOVA: Fixed effects, omnibus, one–way
Number of groups = 4, α err prob = 0.05, Effect size f = 0.4

Plot Parameters

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

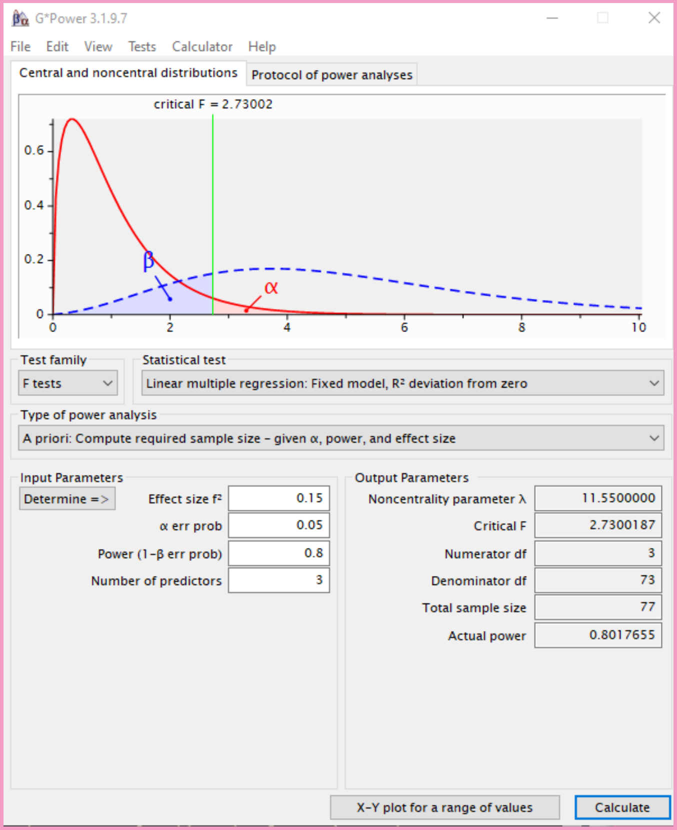
as a function of: Power (1–β err prob) from 0.6 in steps of 0.01 through to 0.95

Plot 1 graph(s) interpolating points

with Effect size f at 0.4

and α err prob at 0.05

Fig. 4. 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

Test family: F tests

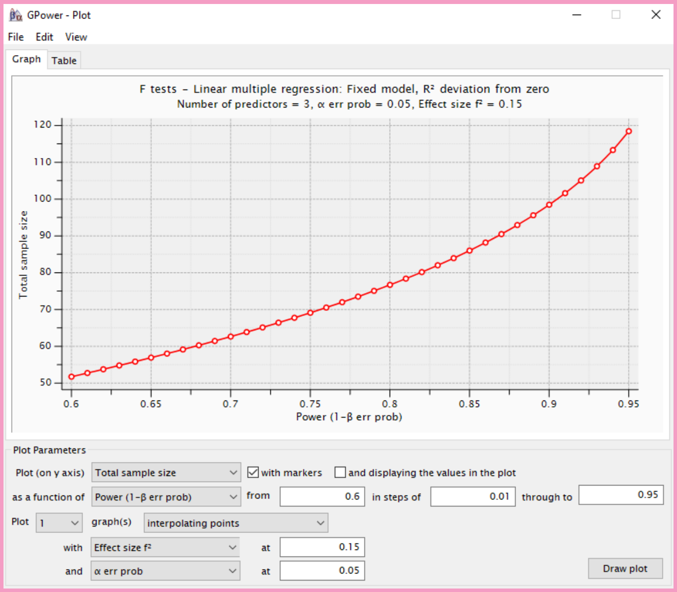
Statistical test: Linear multiple regression: Fixed model, R^2 deviation from zero

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

Input Parameters	
Effect size f^2	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. 5. A Priori Lienar Regression G*Power test

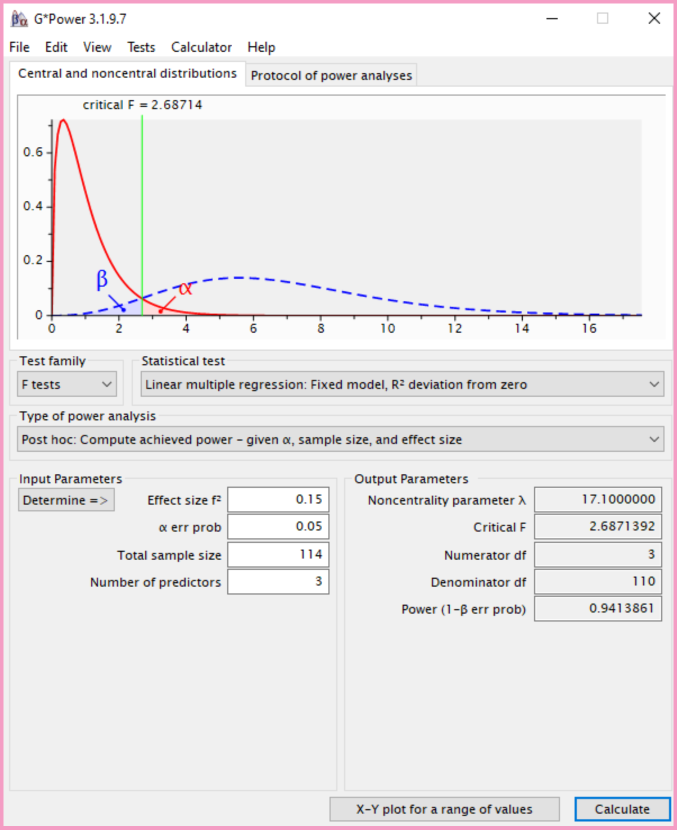


F tests – Linear multiple regression: Fixed model, R² deviation from zero
Number of predictors = 3, α err prob = 0.05, Effect size f² = 0.15

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. 6. A Priori Linear Regression 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 = 2.68714
0.6
0.4
0.2
0
0
2
4
6
8
10
12
14
16
β
α
Determine =>
X-Y plot for a range of values
Calculate

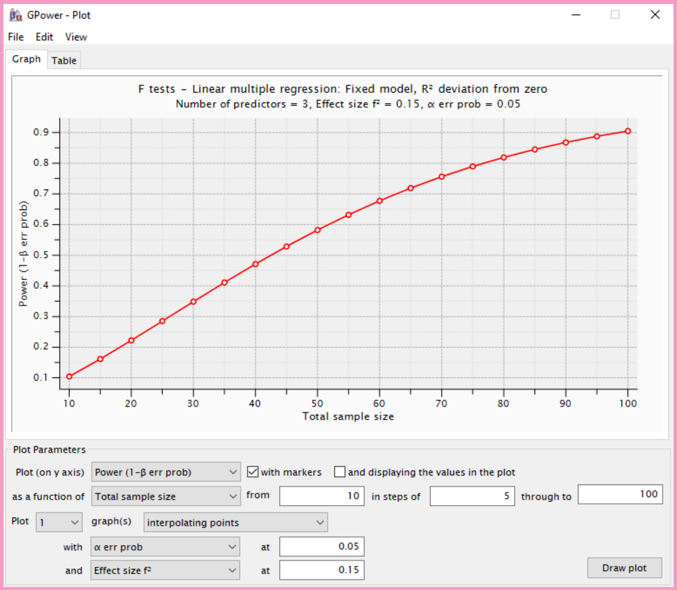
Test family: F tests

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		Output Parameters	
Effect size f²	0.15	Noncentrality parameter λ	17.1000000
α err prob	0.05	Critical F	2.6871392
Total sample size	114	Numerator df	3
Number of predictors	3	Denominator df	110
		Power (1−β err prob)	0.9413861

Fig. 7. Post hoc analysis



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

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. 8. Post hoc graphed

B. 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

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)

  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)
summary(model)


Func_Hypothesis1And2 <- function()
{
  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()
}

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"
    )
}

Func_Hypothesis4 <- function()
{
  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()
}

Func_GetEScore <- function()
{
  defaultValues <- subset(dataFrame, BFC == 0 & FVC
      == 0 & ZC == 0)
  df2 <- dataFrame[-221,]

  #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)
}
Func_ReadFiles()
Func_Hypothesis1And2()
Func_Hypothesis3()
Func_Hypothesis4()
Func_GetEScore()
```

C. Graphs

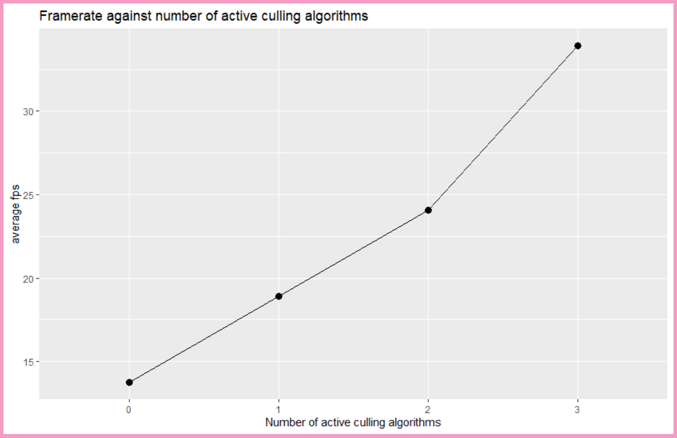


Fig. 9. Framerate against the number of active culling algorithms

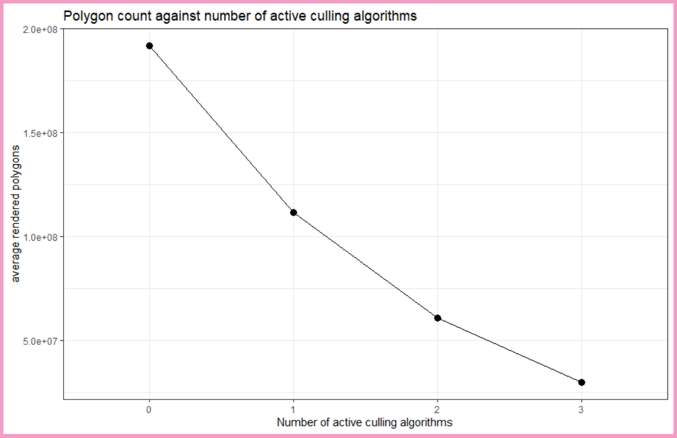


Fig. 11. Polygon count against number of active culling algorithms

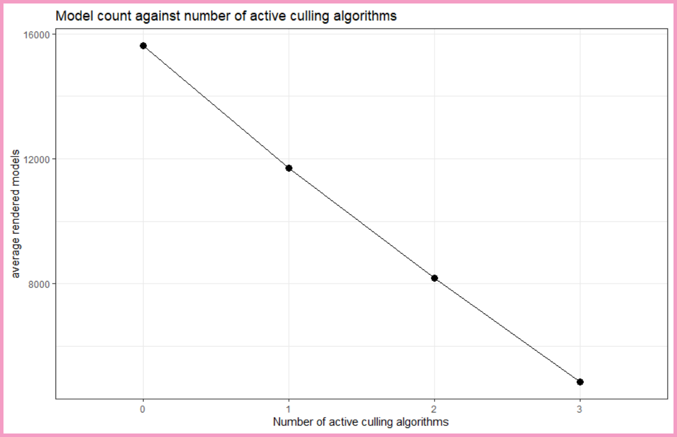


Fig. 10. Model count against number of active culling algorithms

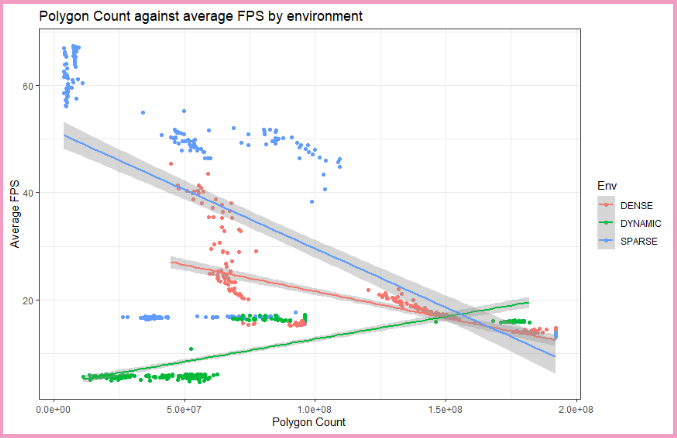


Fig. 12. Average rendered polygon count in each environment

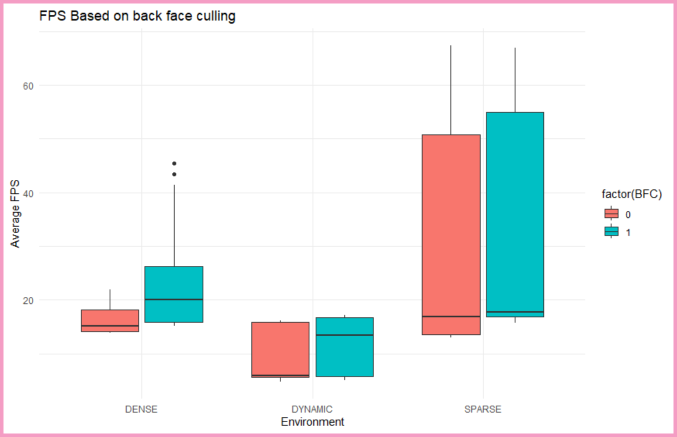


Fig. 13. FPS based on back face culling

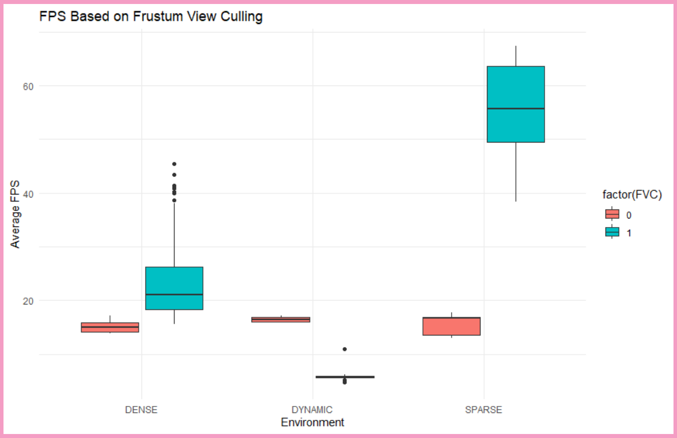


Fig. 14. FPS based on frustum view culling

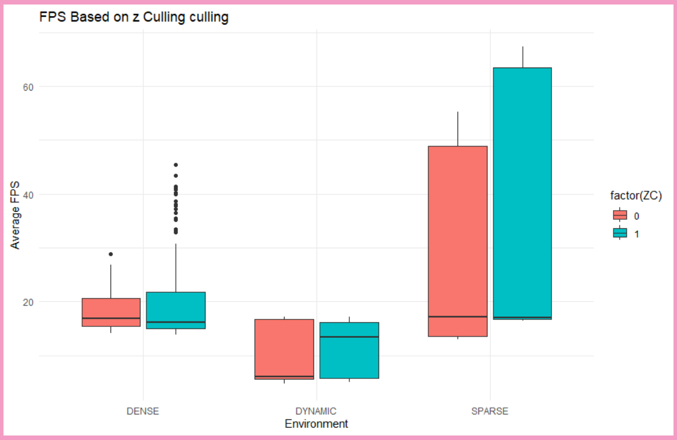


Fig. 15. FPS based on Z Culling