MASTER'S THESIS 2024

Exploring Cloud Rendering Techniques for Aerospace Applications

Amjad Bakir, André Frisk

ISSN 1650-2884
LU-CS-EX: 2024-36

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-36

Exploring Cloud Rendering Techniques for Aerospace Applications

Utforskning av molnrenderingstekniker för flyg- och rymdapplikationer

Amjad Bakir, André Frisk

Exploring Cloud Rendering Techniques for Aerospace Applications

Amjad Bakir
am0108ba-s@student.lu.se

André Frisk
an8218fr-s@student.lu.se

June 24, 2024

Master's thesis work carried out at Tactel Aktiebolag.

Supervisors: Michael Doggett, michael.doggett@cs.lth.se
Magnus Wihlborg, magnus.wihlborg@tactel.se

Examiner: Jacek Malec, Jacek.Malec@cs.lth.se

Abstract

This master's thesis investigates cloud rendering using billboards for aerospace applications. The focus is on creating realistic and aesthetically pleasing depictions of clouds, while taking the hardware constraints of the in-flight entertainment system into consideration. The study aims to render clouds across various weather conditions, exploring different optimization techniques in order to maintain good performance without sacrificing much of the visual quality of the cloud billboards.

In order to accomplish this, cloud billboards are created using 2D images of 3D cloud models and are placed at different locations in 3D space. Depending on the weather data, different cloud numbers and clustering configurations were used. The clouds were optimized with techniques such as instancing and frustum culling. To improve the cloud visual appearence, methods like phong shading and alpha mixing were used. The testing of the solution was carried on Panasonic monitors to assess CPU and GPU load performance. The findings of this study lay the groundwork for future research that focuses on optimizing billboarding techniques for clouds.

The results show that in-flight entertainment systems are ready for real-time cloud rendering.

Keywords: Tactel, Billboards, OpenGL, Android, Performance, Clouds, Panasonic

Acknowledgements

We would like to thank Tactel for making this master thesis possible. A huge thank you for giving us the support we needed throughout this thesis.

We would also like to thank our supervisor Magnus Wihlborg for his contribution and help on this thesis but also Tobias Leksell for providing expertise and knowledge in the graphics subject to us whenever needed.

This thesis wouldn't be possible without our academic supervisor Michaell Doggett and our examiner Jacek Malec from the department of computer science at LTH, big thanks to both of you.

Also, we would like to extend our sincere appreciation to our families and our partners, Julia and Ammeli, for their steadfast support and understanding throughout the duration of our research. Their encouragement has been a crucial part of our ability to complete this thesis successfully.

Contents

Chapter 1
Introduction

In the aviation industry, it is important to give passengers a good experience regarding the flight and its entertainment. The satisfaction with airlines has increased according to an annual travel survey [6]. However, passengers are still tough to please when it comes to in-flight entertainment. Airlines are challenged to keep up the pace of the new technology and are required to make fast and big steps. One solution to satisfy the passengers more is through the use of in-flight entertainment systems. Passengers are far more likely to have a positive experience with an airline if they are entertained during their flight [6]. Data shows that 45% of passengers who travel with high travel-time flights entertain themselves with e.g. movies on the flight [2], which is an important aspect in having an in-flight entertainment system. Another thing that airlines can add to their in-flight entertainment system is a map showing where the plane is in real-time and possible landmarks in the landscape below.

Tactel AB [1] is set to revolutionize the in-flight experience through the ARC [31] project. Focused on leveraging their advanced airplane monitor technology, ARC aims to elevate visual engagement by integrating dynamic cloud graphics into their maps. The core of this project involves harnessing the potential of ARC to generate captivating cloud visuals in real-time. Passengers will be treated to an unparalleled experience as the monitors dynamically render realistic clouds, enhancing the journey and setting a new standard for in-flight entertainment.

1.1 Problem Statement

Tactel AB is a market leader in in-flight entertainment systems based in Malmö, Sweden. This thesis has been carried out at the department at Tactel, responsible for developing and maintaining the engine for their in-flight entertainment system called ARC. To enhance their product for customers, particularly airplane passengers, new standards and innovations must be considered. Currently the engine does not have any weather elements, e.g. rain and clouds, and that will be their next innovation for the engine.

We believe that if the system is able to adapt the view for the passengers with cloud based weather data, the experience for the passengers will increase greatly. This is due to the fact that if we can simulate something from outside the airplane into the system, the passenger will get a realistic feeling of what happens outside to enhance the user experience and customer satisfaction.

Tobias Leksell is a Software Engineer at Tactel with great experience in computer graphics. He has provided a summary of the problem which we and Tactel have discussed about and based our research questions on:

> “Weather is and always has been quite important for humans, and clouds are a very important part of weather. Thus having actual weather effects present in a map is a rather interesting prospect. Clouds are also a great way to convey distances and speed when moving through the air.
> Monitors used in aerospace are very much industrial type of devices, and usually trade performance for reliability and safety. As such modern rendering solutions do not always apply, or need to be scaled down to fit within the confines. — *Tobias Leksell* ”

1.2 Research Questions

The primary goal of this thesis is to explore the use of the billboarding method in aerospace applications and all the challenges that it entails, utilizing real-time cloud data obtained from weather sources. The research questions have been formulated to address the stated problems provided by the case company. However, care has been taken to make the research appealing to a broader audience.

RQ1 How can billboarding techniques be optimized for cloud rendering to achieve realistic representations from various spatial perspectives, considering current hardware constraints and future possibilities?

RQ2 How can cloud rendering techniques such as billboarding be developed to provide realistic and visually appealing real-time representations of clouds using relevant data?

1.3 Delimitations

In order to match the hardware for the systems used in this thesis at a reasonable level we have decided to not use Ray Tracing or Path Tracing. These two methods are out of the scope of this thesis as the systems which we will test on do not have strong enough hardware to support these methods. Instead we will use a technique that adjusts an object’s orientation so that it faces the target, which in this case is the perspective from camera space.

The clouds have different behaviour and forms regarding the weather condition, e.g rain or thunder. However, in the scope of this thesis we will focus on sunny weather in order to get a higher quality on the standardised clouds.

1.4 Related Work / Previous Work

Below we describe the previous works that we found relevant for our thesis which we will use to improve and gain inspiration towards our goal.

The master thesis by Nilsson [18] was also carried out at Tactel to render clouds for the interactive 3D map ARC in real-time. The rendering algorithm used by Nilsson is a combination of physics-based (based on the underlying physics) and procedural approach (based on noise functions and textures for example). First the scene of the clouds is initialized before runtime with all the processes that do not need to be updated continuously. Then the other part of the rendering is done in real-time and is dependant on environmental factors such as camera position and lighting. The model is based on a particle system where each cloud consists of spherical particles with random positions. The illumination of the clouds is calculated by solving the Light Transport Equation (LTE) with a single scattering source.

Olajos [21] uses volumetric clouds and ray marching algorithm which traces rays from the camera unto the scene. This is done to know which clouds to show and also to calculate the lighting of the cloud particles.

Mattsson [17] uses ray tracing to render clouds and smoke. Ray tracing means following the rays from the light source, to the object and then to the camera. Then it needs to be determined which rays hit the camera and in which order so that the correct ones are shown in the front. The smoke is also represented as a particle system.

Behrendt et al [7] use a set of billboards that dynamically change in order to present a scene of plants. Billboards are generated by using a clustering algorithm on the vertices of the triangles making up an object. A cluster is then represented using an oriented bounding box. The billboards are then rendered using either pre-lighting with a single light source or with a reflection function that uses spherical harmonics. In order to render a big scene, a 3D texture is generated by intersecting a gaussian filter with the geometry. The 3D object is then rendered as slices with 2D textures. Then shell textures are used to represent distant objects.

1.5 Contribution

The thesis aims to enhance Tactel's Arc project, which seeks to revolutionize in-flight entertainment with dynamic cloud graphics. Focused on optimizing cloud rendering techniques, particularly billboarding, the research explores realistic representations from various spatial perspectives. By integrating real-time cloud data from weather sources, the study enhances authenticity and visual engagement. Moreover, the research addresses hardware constraints, developing rendering algorithms tailored to aerospace applications. This facilitates easier deployment and maintenance of high-quality in-flight entertainment systems. Ultimately, the thesis aims to differentiate Tactel in the market by providing passengers with unparalleled visual experiences during flights, setting new standards in cloud rendering for aerospace applications, but also to contribute to the research regarding rendering in computer graphics.

1.6 Distribution of Work

At the start of the thesis we split the work evenly to look up theories and related work that has already been done. Before starting the development we made sure to write the related theory in the report. André took care of rendering and taking photos of the different cloud structures from different angles whilst Amjad took care of the initial phase of the OpenGL code to ensure we have a working environment to code in.

The implementation with OpenGL after the initial phase was split between the two of us for faster implementation of important functionality.

Translating the C++ code to Android and deploying the code to different APKs was Amjad in charge of whilst André prepared test cases and scripts for the data collection and plotting.

In the end of the thesis we split up the report writing for the results and discussion where Amjad focused on the GPU performance and André focused on the CPU load and the method.

Chapter 2
Background

There are different methods to render clouds in general, for example particle systems and volumetric rendering. Billboarding is a method that does not require much resources and can produce realistically looking clouds. This is especially important in situations with real-time rendering requirements and hardware limitations. It is notable that most solutions are designed for a terrestrial viewpoint, neglecting the unique challenges posed by an aerial perspective. The thesis aims to address the need for visually appealing cloud rendering from space, including scenarios where one is above, below or going through the clouds. Additionally, cloud data will be sourced from real-time weather information, adding an authentic dimension to the rendering process, possibly also extending to investigating how to implement the solution on the limited hardware in airplanes.

2.1 Clouds

As there exist many different types and structures of clouds we have decided to focus on four types of clouds, Cumulus, Stratocumulus, Altostratus, and Cirrus for the cloud rendering in the ARC engine. These clouds are the most standardised ones. See Figure 2.1 for an illustration of the different cloud structures.

2.1.1 Cumulus

The cumulus clouds typically have a puffy and cotton-like appearance and are usually the clouds which appear in the sky during sunny weather. The cumulus cloud is generally white or light gray depending on where the sunlight is scattered in the water droplets present in the cloud. Cumulus clouds are found in the lower to middle levels of the Earth's atmosphere, usually below 2,000 meters [41].

Cumulus clouds are individual cloud units, meaning they are often separated from each other and not clustered as other clouds. These clouds form when warm air at the Earth's

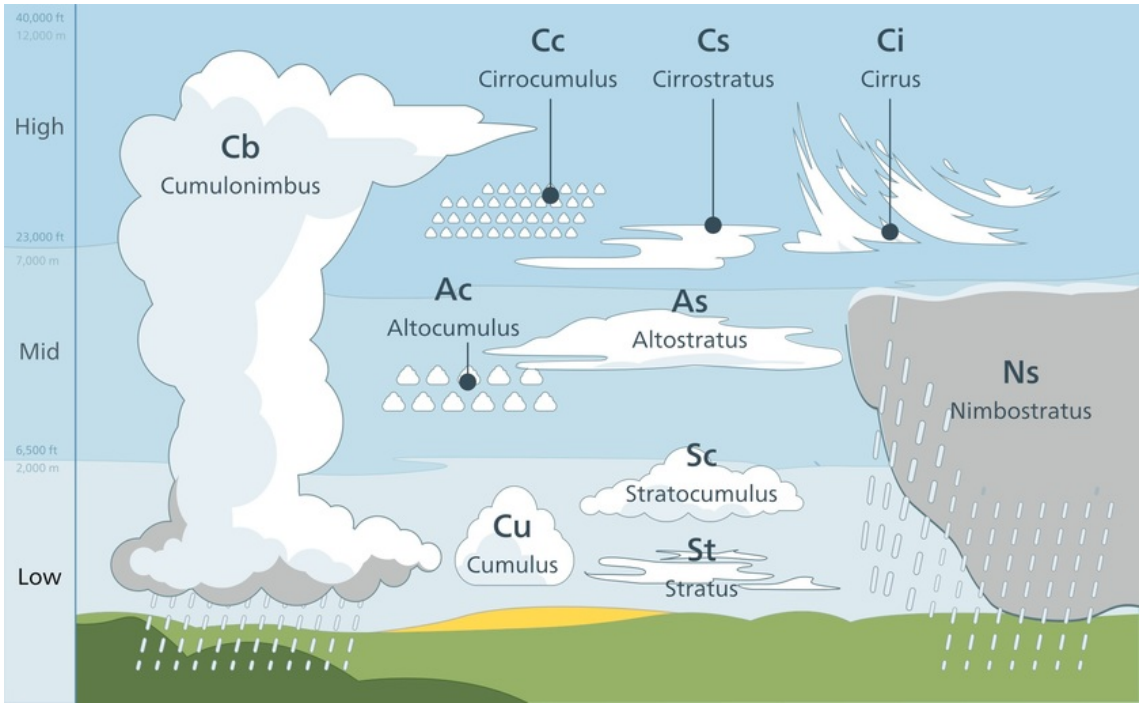


Figure 2.1: Illustration of different cloud structures with altitude range. Source: Science Learning Hub (`https://www.sciencelearn.org.nz/resources/628-observing-clouds-and-weather`)

surface rises and cools as it ascends. As the air cools, it reaches a point where the water vapor in the air condenses into tiny water droplets, forming the visible cloud.

2.1.2 Stratocumulus

The stratocumulus clouds have a layered or stratified appearance and form in a continuous sheet-like layer, often exhibit a lumpy or undulating texture and have a white or light gray color depending on the sunlight scattering [42]. This cloud type can be found at low to middle levels of the Earth's atmosphere, usually below 2,000 meters, and they cover large expanses of the sky and may extend horizontally for great distances. These clouds are often clustered and create very wide cloud formations.

Stratocumulus clouds often form in stable atmospheric conditions when there is a layer of cool, moist air trapped beneath a layer of warmer air. The cooling of the air causes the moisture to condense into cloud droplets, forming the characteristic layered structure. The individual cloud elements, which cover more than 5 degrees of arc each, can connect with each other and are sometimes arranged in a regular pattern.

2.1.3 Altostratus

Altostratus clouds appear as a thin, gray or bluish veil covering the sky. Unlike cumulus or stratocumulus clouds, altostratus clouds lack the distinct individual elements and often

create a uniform layer. The color of altostratus clouds is typically gray or bluish [39]. The thickness of the cloud layer can influence the cloud's color, with thicker clouds appearing darker. These clouds are found at middle level altitudes of the Earth's atmosphere generally between 2,000 to 6,000 meters.

Altostratus clouds are often formed when a layer of moist air is lifted over a front, leading to the condensation of water vapor into cloud droplets and ice crystals. They can also form ahead of a storm system, indicating that significant weather changes may be on the way. Unlike thicker clouds such as nimbostratus, altostratus clouds are usually thin enough to allow the sun or moon to be partially visible through the cloud layer.

2.1.4 Cirrus

Cirrus clouds have a delicate and wispy appearance, often resembling thin, feathery strands or streaks. They can take various shapes, including curls, hooks, and tufts, and their appearance is influenced by high-altitude winds. Cirrus clouds are usually white, but they can sometimes appear to have a golden or reddish tint during sunrise or sunset. The color is a result of sunlight scattering through the ice crystals in the cloud.

Cirrus clouds are among the highest clouds in the atmosphere, forming at altitudes between 4,000 to 20,000 meters [40]. Cirrus clouds form in the upper troposphere where temperatures are extremely cold. They develop from the freezing of supercooled water droplets or directly from the deposition of water vapor into ice crystals. While cirrus clouds are generally associated with fair weather, their presence can indicate changes in the weather. Increasing cirrus clouds may precede the approach of a warm front, and their thickening or transformation into cirrostratus clouds may signal the approach of a storm system.

2.2 The ARC Engine

ARC is a map service developed at Tactel for in-flight entertainment to the passengers. ARC is used for the passengers to track the flight's path with a high-resolution map feature with possibilities to connect other onboard entertainment services to on-the-ground locations and surroundings the flight passes [31]. For example, when passing a city, travelers can both see city maps and learn more about points of interest. They can also get deals on everything from hotels to restaurants and events. The system can for example offer correct prayer times for Muslims where it points to Mecca and many more features.

The map is built using satellite imagery with 3D elements to create realistic images for the system. It reads data from the environment of the airplane to ensure realistic weather conditions and terrain imagery. For example, when the sun goes down, the light on the screen also fades out. ARC offers different views of the airplane with an overview of the flight, the *Above cam* which shows a closer look of what is nearby the airplane and the *Chase cam* which shows the field of view (FOV) from a third-person perspective. Please see Appendix A and Figure 2.2 for more detailed images of the ARC system.

The greatest challenge with the ARC system is to ensure it has a high performance whilst considering hardware constraints. The imagery can not take too long to render or take up too much work on the hardware as it might make the system slow or not work as intended. For reference later, the ARC engine runs on average 47 FPS (frames per second) on the monitors.

Figure 2.2: Overview cam of the ARC in-flight entertainment system. Source: Tactel

2.2.1 Panasonic eX3

The testing in this thesis will be done on three Panasonic monitors which are used by Tactel for testing the software on. The first monitor is the Panasonic eX3 [5]. This monitor is the standardised monitor used in the aviation industry for in-flight entertainment systems hence we will use this model. The Panasonic eX3 has the following specifications:

- **i.MX 6Quad Processor** [28] - 4x Arm® Cortex®-A9 up to 1.2 GHz per core, 1 MB L2 cache, 32 KB instruction and data caches, NEON SIMD media accelerator
- 4 GB RAM
- Integrated GPU in the CPU - **Vivante GC2000** [10] with 700MHz clock speed, 4 (Vec-4) and 16 (Vec-1) shader cores and shader GLOPS around 25 (High) and 50 (Medium)

The i.MX 6Quad processor is a system-on-chip (SoC) designed by NXP Semiconductors [28], commonly utilized in various applications including computer graphics. It features a quad-core ARM Cortex-A9 CPU architecture, offering high performance and power efficiency. With its advanced graphics processing unit (GPU) capabilities, it supports smooth rendering of complex visual content, making it suitable for tasks ranging from multimedia playback to gaming.

2.2.2 Panasonic NEXT

The second monitor is the Panasonic NEXT [4] which is a newer version of the Panasonic eX3 (at the test site at Tactel). It has older hardware but is still used in the industry for a more budget friendly monitor for in-flight entertainment systems. The Panasonic NEXT monitor has the following specifications:

- **Qualcomm Technologies Inc APQ8096pro** [29] - Clock speed 2.3 GHz, 4 Cores, 4 threads
- 4 GB RAM

- Integrated GPU in the CPU - **Adreno 530 GPU** [19, 33] with turbo speed of 620 MHz

The APQ8096pro processor is a SoC designed by Qualcomm Technologies [32, 33] and supports 4K Ultra HD video capture and playback, as well as advanced audio features such as Qualcomm Acoustic audio codec and support for Hi-Fi audio.

2.2.3 Panasonic Astrova

The third monitor is the Panasonic Astrova [3] which is a new model from Panasonic to be used in in-flight entertainment systems. It has better hardware, newer software and better performance but uses similar hardware as the NEXT monitor. The Panasonic Astrova has the following specifications:

- **Qualcomm Technologies Inc APQ8096pro** [29] - Clock speed 2.3 GHz, 4 Cores, 4 threads
- 8 GB RAM
- Integrated GPU in the CPU - **Adreno 530 GPU** [19, 33] with turbo speed of 620 MHz

2.3 The Weather Data

The data for the ARC engine regarding weather comes from the Foreca API. This API provides weather data from various locations which is marked by an ID. This ID can be looked up in the Geonames database to get information regarding which city the data comes from. This data comes both hourly and daily to ensure updated data for the product. The following example snippet contains the structure of the data being sent to ARC hourly, in this case from the city of Lahore in Pakistan.

```
"id": "1172451",
"weather": [
    {
        "s": "d000",
        "t": "2024-02-05T08:00Z",
        "temp": 16,
        "wd": 343,
        "ws": 3
    },
    ...
```

From the data we can see that Lahore which has the ID *1172451* with the Zulu time of *08:00* have a temperature of 16 degrees Celsius with other properties such as wind direction in degrees and wind speed. The most important parameter in this schema is the *s* parameter indicating the weather symbol code, in this case *d000*. Foreca has a sheet for what every weather symbol code means, we have provided a short part of the sheet, see Figure 2.3. Weather symbols for the night also exist.

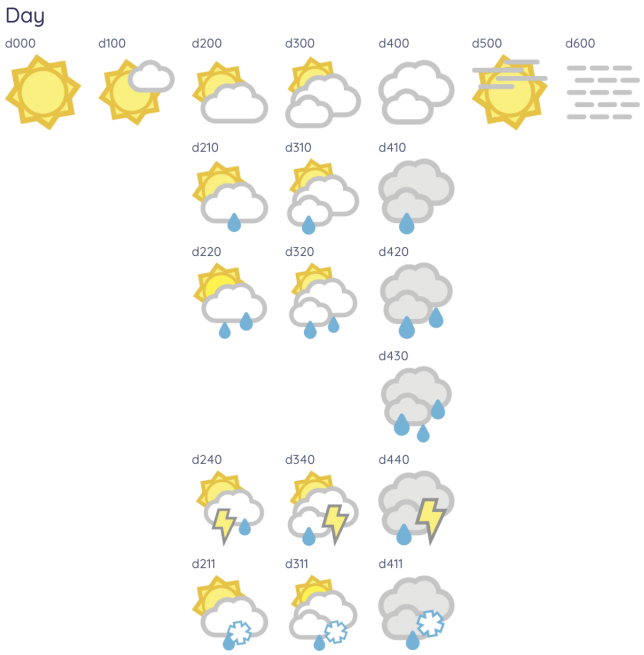


Figure 2.3: One part of the Foreca weather symbol code sheet. Source: Foreca (`https://developer.foreca.com/resources`)

Here we can see that the weather symbol code means that we will have very sunny weather with zero clouds in the sky. This will be useful for the ARC engine to determine which cloud to render based on this data.

The daily data however looks a bit different and is used for other purposes to the ARC engine as it only provides with maximum and minimum temperature of the day with the mean weather symbol code.

```
"id": "1040652",
    "weather": [
        {
            "d": "2023-09-05",
            "maxT": 33,
            "minT": 18,
            "s": "d100"
        },
        ...
```

This can be useful for future implementations if the passengers can themselves decide if they want to show the mean weather or the hourly weather.

2.4 Theory

In this section we will explain the different theories we will apply in the thesis. This includes the billboarding method, optimizations such as frustum culling, how lighting works etc.

2.4.1 The Billboarding Method

Billboarding is a technique that uses 2D elements or sprites in a 3D world [24]. A sprite is a common item used in computer graphics that refers to a 2D image or animation that is integrated into a larger scene or in a game environment [15] where they can represent characters, objects or special effects. Sprites are used to create dynamic and visually appealing graphics where they can give an illusion of movement or interaction, which is what we will use in this thesis. Rectangles with texture that are reshaped so they always appear parallel to the view plane are known as billboards [38]. They are therefore rotated for optimal visibility, much like billboards beside highways. They differ from highway billboards, though, in that they rotate dynamically to provide optimal visibility at all times.

Billboarding can be an efficient way to render a large number of complex objects since it uses minimal geometry and does not need as much memory.

In Billboarding a surface normal and an up direction are found for orienting the quadrilateral. These two vectors are required to create an orthonormal basis for the surface [34]. What these vectors tell us is the rotation that the matrix needs to rotate the quadrilateral to its direction. Then, an anchor location is required on the quadrilateral to manifest the position in space. The desired surface normal $\boldsymbol{n}$ and up vector $\boldsymbol{u}$ are often not perpendicular. To align the quadrilateral, one of these vectors is chosen as a fixed vector, maintaining its direction. The other vector is then made perpendicular to the fixed vector through a consistent process. This involves creating a "right" vector $\boldsymbol{r}$, pointing to the right edge of the quadrilateral, achieved by taking the cross product of $\boldsymbol{u}$ and $\boldsymbol{n}$. The resulting vector $\boldsymbol{r}$ is normalized and used as an axis in the orthonormal basis for the rotation matrix. See Figure2.4 for visualization of the vectors. Further the vector of $\boldsymbol{n}$ or $\boldsymbol{u}$ which is not fixed will be adjusted by taking the cross product of this fixed vector with $\boldsymbol{r}$ which results in a vector perpendicular to both these vectors. If the normal $\boldsymbol{n}$ is fixed then the new vector $\boldsymbol{u}'$ is:

$$u' = n \times r,$$

whilst if the up vector $\boldsymbol{u}$ is fixed then the new vector $\boldsymbol{n}'$ is:

$$n' = r \times u.$$

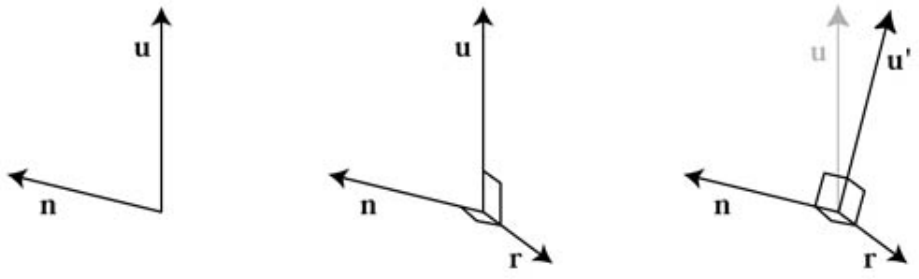


Figure 2.4: Demonstration of acquiring the axis in the orthonormal basis. Source: Flipcode, Tomas Akenine-Möller and Eric Haines (`https://www.flipcode.com/archives/billxfrm.jpg`)

The new vector is then normalized and the three vectors are used to form a rotation matrix. For example, for a fixed normal n and adjusted up vector u' the matrix is:

$$M = (r, u', n).$$

A quadrilateral in the xy plane, with +y denoting the upper edge and the quadrilateral centered around its anchor location, is transformed by the matrix. It is oriented correctly thanks to this modification. The quadrilateral's anchor point is then moved to the desired location using a translation matrix.

We will be using screen-aligned billboards. In computer graphics, screen-aligned billboards, also referred to as camera-facing billboards, are a popular method for generating objects, like sprites or textures, that always face the camera [34]. Rather than following the global coordinate system, these billboards are oriented according to the screen space. Because of this, they always remain oriented in relation to the viewer's perspective, irrespective of the viewer's movement or position within the scene. They offer a practical means of displaying 2D graphics in a 3D space while guaranteeing that they are always facing the user or camera.

2.4.2 Blending

Blending is the stage of OpenGL rendering pipeline that takes the fragment color outputs from the Fragment Shader and combines them with the colors in the color buffers that these outputs map to [13]. It is in other words a tool for determining the correct color of objects which can be used if the objects are close to each other. For blending of the clouds the correct blending technique for realistic attributes to use is *Alpha Blending*. Alpha blending is a technique used in computer graphics to combine two images or objects by specifying a transparency value for each pixel of one image or object, known as the alpha channel. This allows for smooth and realistic blending of images, such as overlaying a semi-transparent object on top of another. Each pixel in an image have an associated alpha value ranging from 0 to 1, where 0 means invisible or fully transparent and 1 is fully opaque. When blending pixels together, an alpha blending equation can be used:

$$ResultingPixelColor = SourcePixelColor \times SourceAlpha + DestinationPixelColor \times (1 - SourceAlpha)$$

This equation combines the colors of the source pixel and the destination pixel based on their alpha values. This ensures that transparency elements in the scene will be visible and we can guarantee that, e.g. clouds can be seen through so we can see what is behind the cloud (depending on the alpha value).

Alpha Blending is not all that is required in order to make transparency work. If we have two clouds in front of the camera, and the cloud closest to the camera view is rendered first and then the other cloud is rendered behind the first cloud, the rendering could go wrong by having overlapping textures or blending which does not work properly. In other words, the cloud behind might not be shown if the front cloud has transparency elements. This can be fixed by using the *back-to-front rendering* or *depth rendering* [13]. This technique ensures that the objects which are furthest away from the camera view will be rendered first, and the remaining objects will be rendered based on the furthest distance from the camera view. By doing this transparency and blending will work as intended. Why this works is because of the Alpha Blending algorithm, as we need the destination pixel in order to calculate the resulting pixel color, but if we do *front-to-back* rendering (which is the most normal way of

rendering), the destination pixel color might not be rendered in time resulting in incorrect transparency. However, this technique does not take angle into account, which can or might be an obstacle with Billboards.

As an alternative to *back-to-front rendering*, we have experimented with stochastic transparency [9] which does not need depth sorting. Instead, this method discards pixels by comparing the pixel's alpha value to a random threshold, if it is below the threshold then the pixel is discarded. When sampled multiple times, the average of the visible fragments creates a transparency illusion. The problem with this method is that it introduces noise, worsening the visual quality of the clouds while also needing to be sampled multiple times each frame so we have decided to focus entirely on the alpha blending algorithm mentioned above.

2.4.3 Frustum Culling

Frustum Culling is a technique used in computer graphics for improving rendering performance by eliminating objects which do not appear in the viewing frustum [11]. This means that objects outside of the camera's view will not be rendered. Most commonly, this will be the objects behind the camera. Please see Figure 2.5 for an illustration of how it works, where the gray objects are inside the view frustum.

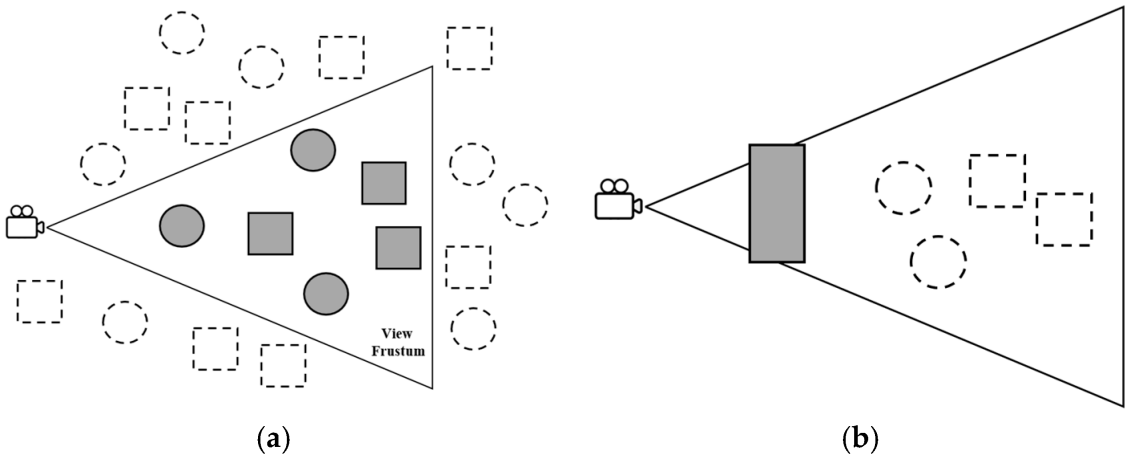


Figure 2.5: Visualization of how Frustum Culling works from the camera view. Source: Lee, E.-S.; Shin, B.-S. Vertex Chunk-Based Object Culling Method for Real-Time Rendering in Metaverse. Electronics 2023, 12, 2601. (`https://doi.org/10.3390/electronics12122601`)

We define a frustum which is the camera's position, view direction, field of view angle and aspect ratio. In other words, it defines the camera's perspective. Then, for each object in the scene we check if the bounding volume of the object, e.g. bounding box or bounding sphere, is inside the frustum. If any part of the bounding volume is in the frustum then the object will be visible. If the object's bounding volume is not inside the frustum, the object is culled, meaning that it is not rendered, increasing the performance as it reduces the number of objects that the rendering pipeline must process. By eliminating objects that are not visible

to the camera, the graphics processing unit (GPU) can focus its resources on rendering only the objects that contribute to the final image, resulting in improved frame rates and overall performance.

2.4.4 Rayleigh scattering

Rayleigh scattering occurs due to the interaction of light with small molecules in the atmosphere. Rayleigh scattering refers to the scattering of light by particles in its path of size up to one-tenth the wavelength of the light and occurs without any loss of energy or change of wavelength [16]. This phenomenon results in a more pronounced scattering of shorter wavelengths, with blue light being scattered most significantly, followed by green and red [22, 16]. The blue color of the sky is attributed to the fact that blue light scatters in various directions, ultimately reaching our eyes from all angles. During sunset, the sun's light appears yellow, orange, or red because as it traverses the atmosphere over long distances, much of the blue and a considerable amount of green light are scattered away, leaving predominantly reddish hues.

2.4.5 Mie scattering

Mie scattering refers to the elastic scattering of light by particles whose diameters are comparable to or larger than the wavelength of the incident light [12]. The intensity of the Mie signal is directly proportional to the square of the particle diameter. In comparison to Rayleigh scattering, Mie scattering is significantly more potent, making it a potential source of interference for Rayleigh scattering, which is a weaker light scattering process. Mie scattering is induced by larger particles in the air, such as aerosols (e.g., dust and pollution). Unlike Rayleigh scattering, Mie scattering tends to scatter all wavelengths of light uniformly [22]. On hazy days, Mie scattering contributes to a somewhat gray appearance of the sky and results in a prominent white halo around the sun. Additionally, Mie scattering can be utilized to simulate the scattering of light by small water and ice particles in the atmosphere.

2.4.6 Sunlight scattering in clouds

The color of a cloud depends on the light the cloud receives from the sun. Clouds typically exhibit a white appearance, but various atmospheric factors can alter their color. To get the color of a cloud which mostly is white or gray we need to make use of Rayleigh scattering and Mie scattering. These techniques are often used in the gaming industry for rendering of light from the perspective of the player.

The color of the clouds is the result of a combination of the Rayleigh and Mie scattering. Unlike Rayleigh scattering, which occurs when light waves are larger than the gas molecules in the atmosphere, Mie scattering takes place in clouds where individual water droplets are of similar size to the wavelength of sunlight. Mie scattering uniformly scatters all colors, resulting in perceived white light. However, clouds may exhibit colors other than white due to atmospheric factors. Thickening clouds may block or diminish sunlight, giving them a

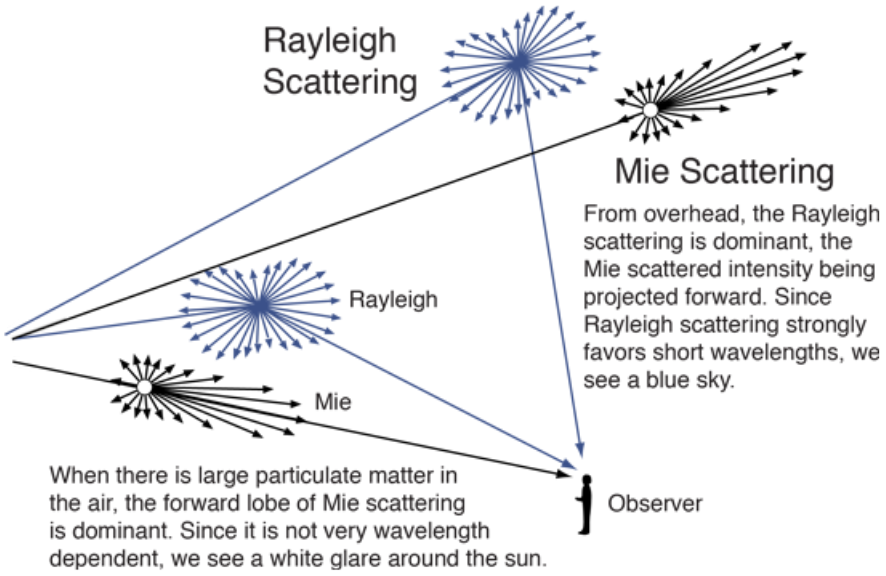


Figure 2.6: Visualization of how Rayleigh and Mie scattering work as seen by an observer in graphics. Source: HyperPhysics, hosted by the Department of Physics and Astronomy at Georgia State University (`http://hyperphysics.phy-astr.gsu.edu/hbase/atmos/imgatm/raymie.png`)

gray appearance. In the absence of direct sunlight, clouds may reflect the color of the sky, appearing bluish. For further visualization of this please see Figure 2.6.

During sunrise and sunset, the Sun's position is low on the horizon, leading its rays to traverse the Earth's atmosphere in proximity to the densest regions. The inclined entry of light at this juncture results in heightened refraction due to the increased atmospheric density, causing the light to take a longer path and facilitating more Rayleigh scattering [20]. In this longer path most of the shorter blue wavelengths are scattered in this path leaving the majority of longer waves to continue. When the light continues to move through the atmosphere the yellow wavelengths become orange wavelengths when scattered, and in the end the orange wavelengths become red wavelength which provides the default color of the sun in our eyes. In other words, near sunrise and sunset the color of the cloud is determined by whatever sunlight color it receives after Rayleigh scattering [20] where Mie scattering scatters the remaining wavelength colors equally within the cloud.

2.4.7 Using the sunlight to determine the cloud color in the ARC engine

A technique used in the ARC engine for the sunlight is called *Raycasting* and is very common for simulating lighting in graphics. This technique works by simulating light rays from the eye of the beholder. Every pixel on the monitor acts as a ray of light, which is estimated to determine how much light falls on the point of the beholder [36]. By doing this we can assume that the light has an intensity and a color depending on the location of the pixel. This takes up less resources as it only requires to render *1920x1080* pixels instead of calculating billions

of pixels outside the eye of the beholder.

In terms of cloud coloring in the ARC engine which is a 3D world, we need to know for every point in this 3D world how much light intensity and which color the light has in this point. This is important to determine the coloring, but we also need to know the angle from the light source to the object in question. This is because the color is determined based on the angle at which the light hits the cloud [20].

In order to get the angle for determining the cloud coloring we will use the dot product:

$$\theta = \arccos\left[\frac{a \cdot b}{|a||b|}\right].$$

We need two vectors which will be the vector from the sun to the cloud which is a unit vector and then a vector from the cloud to the camera. This will give us the angle where the lighting intensity of the cloud is the greatest, while the remaining cloud coloring depends on the direction from which we are viewing it. The lighting however depends on if the cloud is above, below or in the same altitude as the camera. If we take the example of a cloud being above the camera, then the cloud from the point of view appears to be darker as the sun's light is mostly at the top of the cloud closest to the sun.

2.4.8 Microsoft Flight Simulator 2004

The cloud rendering in the game Microsoft Flight Simulator 2004 works quite similar to the idea we have to implement cloud rendering in the ARC engine. This game uses the Billboarding method as previously discussed in order to make the clouds look real and authentic in the game. By locking the facing angle of the sprites, when the camera is within half of the sprite's radius, the clouds would not change their angle relative to the camera, as it is too close [37]. Whilst the camera is outside this sprite radius the clouds will change their facing angle towards the camera in order to create an illusion of that we have 3D clouds.

The game uses a spatial partitioning algorithm to determine cloud frequency by dividing the sky into a 3D grid of cells [37]. Each cell represents a "box," and the algorithm processes meteorological data and applies noise functions to simulate natural cloud variations within these boxes. These boxes exist in sections of different sizes, for example a 16 square kilometer section of clouds can contain 20-200 boxes where each box have 1 up to 100 clouds in it depending on how dense it should be. This is something we need in the scope of this thesis to determine how much clouds should be visualized in terms of cloud frequency.

Chapter 3
Approach

In this chapter, the approach used to answer the research questions is explained. This involves the experiment setup, the infrastructure used in order to perform these experiments and the research methods used such as Frustum Culling and Blending.

The research adopted a multi-faceted approach, involving both theoretical investigations and practical implementations. This includes an in-depth study of existing cloud rendering techniques, such as billboarding, with a focus on their applicability to aerial perspectives and the challenges accompanied with that. The development of novel rendering algorithms will be explored, keeping in mind the unique requirements of rendering clouds for aerospace scenarios. Real-time cloud data from weather sources will be integrated into the rendering process, enhancing the visual representation.

3.1 Generating clouds with Blender

The images of clouds or blender packages which had premade clouds online were not good enough. We wanted to capture the sprites of the clouds ourselves and not download anything already done. At first, we tried to create our own clouds from scratch which was a very difficult task as neither of us have any design experience. After some research we found a package [23] in the Blender community which included all the different cloud formations and structures that exist with an average of six different models of each cloud, giving a high variety for capturing sprites.

For the different cloud structures which we will use in this thesis we used three different models per cloud structure which we captured in different angles for better variation during rendering. For every model we captured:

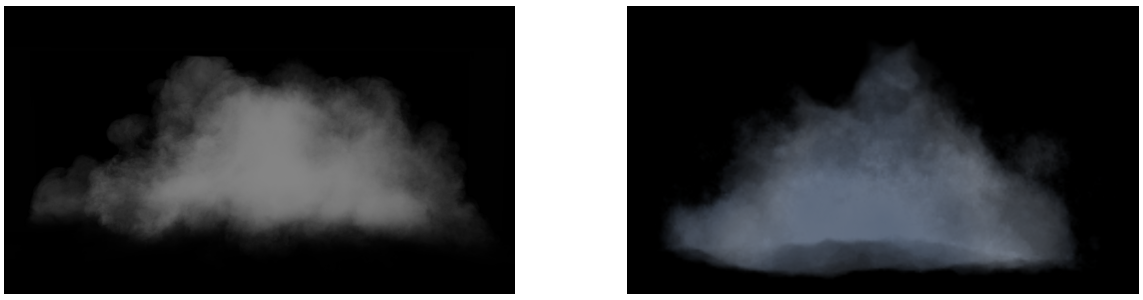
- Cumulus - 5 images on each cloud structure
- Stratocumulus - 5 images on each cloud structure

- Altostratus - 3 images on each cloud structure with sprites focused on the angle of looking at it from a 45 degree angle or from below or above it

- Cirrus - same as altostratus

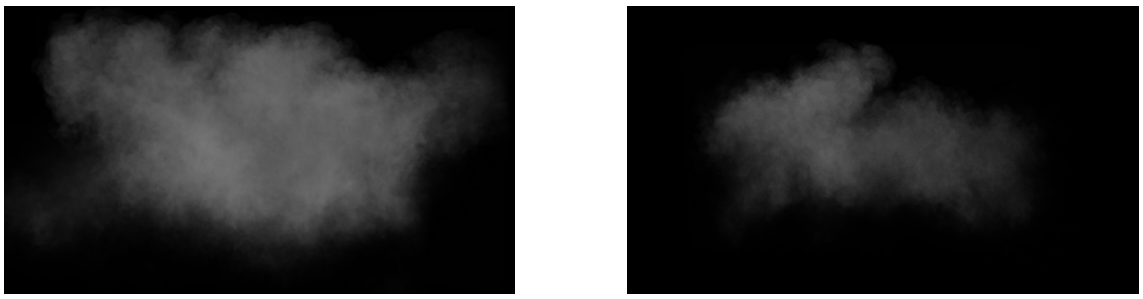
This resulted in 15 images for cumulus and stratocumulus and 9 images for altostratus and cirrus respectively. Please see Figures 3.1, 3.2 and 3.3 for examples of the captured images. We needed to capture more images for cumulus and stratocumulus because these two clouds can differ quite much in terms of how they look as opposed to altostratus and cirrus which have a quite standardised appearance. Also, cirrus and stratocumulus will be the clouds we will use the most in the simulations due to their frequent appearance in weather. In terms of angles we focused on capturing images from below, above, from the front and from a 45 degree perspective so that we check all the different possibilities from the camera view. This is important as the sprites will appear as 2D objects and not 3D, hence the requirement of capturing these angles.

These clouds are then indexed in the code as arrays.



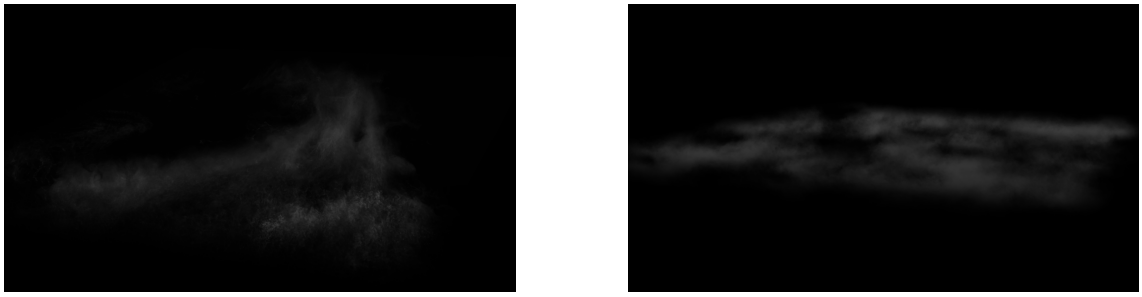
(a) **(b)**

Figure 3.1: Example snapshots of cumulus



(a) **(b)**

Figure 3.2: Example snapshots of stratocumulus



(a) Altostratus **(b)** Cirrus

Figure 3.3: Example snapshots of altostratus and cirrus

3.2 Building the project as an Android application

The ARC engine is a complicated codebase with multiple different components and tests to ensure a good quality code. The engine is complicated and the team had very limited time to help with inserting our own codebase into the engine. Because of this we decided together with our supervisors to build a stand-alone Android application to test on the Panasonic monitors instead. This only affects Tactel in terms of implementing clouds directly in the engine, however the performance testing will be the same as if we would do it in the engine and the research questions stay the same. Hence, this is the best solution for this case.

We used our existing C++ code files and in order to be able to run it on Android, we used Android NDK (Native Development Kit). NDK facilitates the integration of native C++ code into Android applications.

In Java, we used GLSurfaceViewRenderer where specific methods which deal with the initialization of buffers and OpenGL draw calls were declared as native, indicating that their implementations would reside in C++ rather than Java.

Our native methods were implemented in a shared library (.so file) that was created by compiling the C++ code using the Android NDK tools. Next, we included this shared library to the package that contained our Android application.

JNI (Java Native Interface) served as the bridge between Java and native code, enabling communication and data exchange between the two. This allowed us to call native methods from our Java code and pass data seamlessly between Java and C++.

3.2.1 Deploying to the Panasonic monitors

When the Android application builds were finished, we got an APK file which can run on the monitors. In order to transfer the APK and use debugging we used the ADB (Android Debug Bridge) [8]. ADB is a command-line tool that allows developers to communicate with an Android device from a computer. It enables various actions like installing and debugging apps, accessing the device's file system, and executing shell commands. ADB is used in the command line interface (CLI) to transfer the app. Please see Appendix B.6 for instructions of how to install the application with ADB.

3.3 Implementation

In this section we will explain the different parts of the implementation to optimize for performance and build a realistic simulation of clouds.

3.3.1 Cloud positions

After reading the weather condition and retrieving the weather code we generate a number of clouds based on that weather code. The cloud positions are randomly generated within a lower and upper bound on the y-axis depending on which cloud type it is where cumulus and stratocumulus render on the lowest altitude, altostratus renders a bit higher and cirrus in the highest altitude. The amount of clouds generated is based on the weather data gathered from Geolocations where the weather code comes from Foreca. However, as this implementation is in a stand-alone application and not in the engine as mentioned before, the weather data is instead randomised but acts the same as we would use the Foreca API to retrieve the weather code. In the following table we can see how our implementation determines the amount of clouds based on the weather code with a hard coded amount to be rendered in hard brackets:

- D000: No clouds (sunny) [0 clouds]
- D100: Few clouds (sunny with some occasional clouds) [10 clouds]
- D200: Some clouds (sunny with a good amount of clouds) [30 clouds]
- D300: Many clouds (clouds cover a large amount of the sky) [75 clouds]
- D400: Very cloudy (clustered clouds which cover the majority of the sky) [200 clouds]

These weather codes determine how close the clouds should be to each other as for example weather code D100 has 10 clouds. These clouds should be positioned with a certain distance from each other to spread them out to get a more realistic simulation. We apply this if the cloud is of cloud type cumulus or stratocumulus based on the weather code as altostratus and cirrus are more stand-alone clouds and will not cluster. Also, altostratus and cirrus are very wide clouds so it would not make sense to cluster. See the code for determining the cloud positions in Appendix B.2 for a more technical explanation.

In order to mimic the weather from the reality we have a function which adjusts the positions of clouds based on specified weather conditions. The function requires an array of cloud positions, the number of clouds, and the current weather type as parameters. If the weather code is *D100* or *D200*, it calculates the minimum distance allowed between clouds (set to a constant unit value) and iterates through each pair of clouds. If the distance between two clouds is less than the minimum allowed distance, it adjusts the position of one of the clouds to create more distance between them. This adjustment is done by moving the cloud along the direction vector towards the other cloud, ensuring a separation equal to the minimum allowed distance. In these weather codes the cumulus and stratocumulus clouds are often not clustered which is why this function is used to gain distance between the clouds to create a realistic view of the mentioned weather codes. See the code in Appendix B.3 for a more technical explanation.

3.3.2 Instancing

To gain a performance boost, we decided to experiment with instancing. Instancing in computer graphics refers to the technique of rendering multiple copies of the same object (or group of objects) in a scene with a single draw call [14]. Instead of issuing separate draw calls for each instance of an object, instancing allows for efficient rendering by reusing the geometry data and minimizing CPU to GPU communication overhead. Each instance can have its own transformation (position, scale & rotation) applied to it, allowing for variation and complexity in the scene while maintaining performance efficiency. By using instancing we loaded the different textures in a texture array, created a buffer containing the model matrices associated with the different clouds and a buffer for texture indices where each cloud is associated with an index that depends on the type of cloud it is. We then used instancing to draw all the clouds in one call. Please see Figure 3.4 for an illustration of instancing, where the upper part of the image is without instancing and the lower part of the image is with instancing.

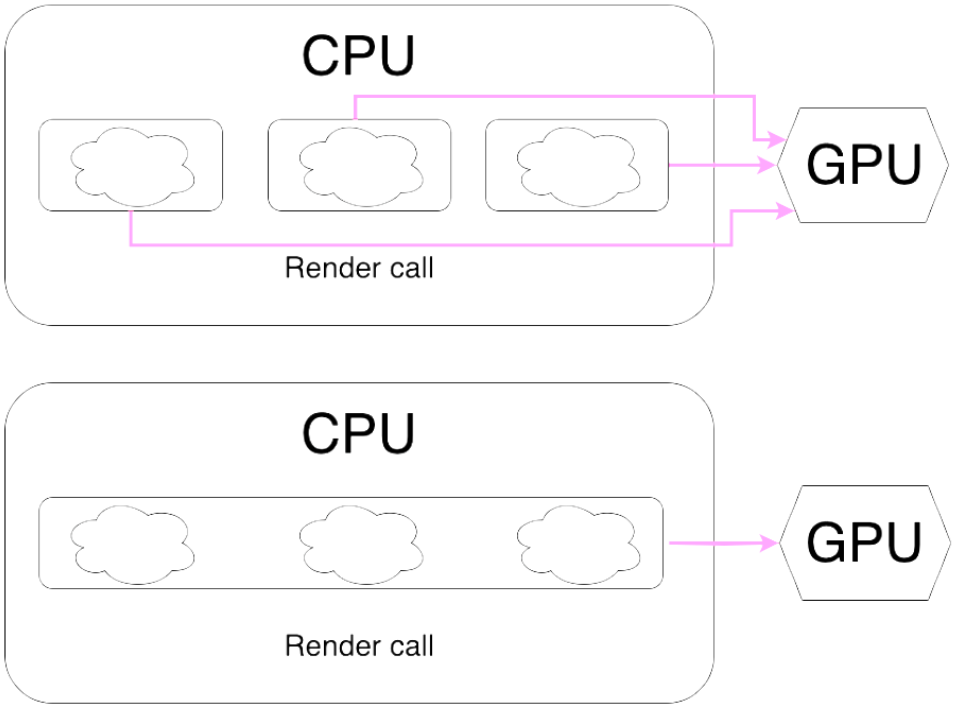


Figure 3.4: Visualization of how Instancing works. Upper image is without instancing and lower image is with instancing. Source: Frisk, André. 2024

3.3.3 The Cirrus implementation

The implementation for the cirrus cloud will always appear above the camera, as it generally appears in altitudes higher than the commercial airplane fly height. In order to make it more

realistic with cirrus we decided to not use billboards for this cloud type after some failed tests with billboards where the change in angle perpendicular to the camera breaks the 3D illusion. Instead we enlarge the cloud to cover a large proportion of the sky. Cirrus is a large cloud type and requires these properties in the engine to give the passengers a better experience of the IES (In-flight Entertainment System). The enlarged cloud together with an altitude which is higher than the simulated airplane will give a realistic representation as the cloud will create an illusion of a 3D object with this distance and size. The maximum amount of cirrus clouds to be active during a simulation is two due to the broad size of it and for a more realistic simulation.

3.3.4 The Altostratus implementation

The implementation for altostratus is more or less the same as for cirrus with the exception that this cloud appears just above the altitude of cumulus and stratocumulus. This cloud engulfs the sky when active and can be flown through as the camera or airplane will be above this altitude. The cloud however will always be flat and does not use billboarding to give a better simulation in the IES. The choice not to use billboards for altostratus is the same as for cirrus. The maximum amount of altostratus clouds also corresponds to two due to the size and realistic simulation.

3.3.5 Frustum Culling behaviour

The implementation for frustum culling in this thesis is a little bit different from the theory implementation to save on resources. Instead of removing the object or hide it from rendering, we instead check if the object is outside of the camera view or frustum and relocate the cloud within the new camera view position further ahead if it is outside the frustum. This is in order to gain a more realistic movement and rendering of the clouds as the camera moves forward or sideways then there will always be new clouds to visualise by relocating them. Instead of rendering thousands of objects we can simply render e.g. for cloudy weather around 50 clouds and stick to this amount to gain better performance as it does not require a huge amount of objects to render and lowers the CPU to GPU communication. The code in Appendix B.1 applies the frustum culling technique and relocates the clouds.

The important part here is the dot product between the camera front vector and the distance from the cloud to the camera's position. This dot product calculation gives a value in the range -1 to 1 where if the dot product is greater than 0, the two vectors are pointing roughly in the same direction and if the dot product is less than 0, the two vectors are pointing away from each other [26]. If the dot product is less than zero, the cloud is behind the camera and thus outside the frustum and needs to be relocated. The code also checks if the distance to the cloud is greater than 2.0 units to ensure that the cloud is not only behind the camera but also sufficiently far away, with the distance threshold preventing clouds very close to the camera (within 2.0 units) from being considered for further processing and ensuring they are relocated.

The cloud type determines the maximum height the cloud is able to be located at. The clouds are being moved relative to the camera position, with variations in their coordinates (x, y, z) based on random offsets and the direction faced by the camera. For altostratus and cirrus the new Y coordinate is calculated by adding the move distance, which is a randomised

distance, to the current Y coordinate of the cloud's position. This approach effectively moves the cloud along the camera's view direction while preserving its height. The new X and Z coordinates are determined by multiplying the camera's front vector by the move distance. For each cloud type, the maximum height for that cloud type is considered when updating the Y coordinate. The `min` function ensures that the new Y coordinate does not exceed this maximum height keeping the natural height possible in the simulation. We update the location of altostratus and cirrus clouds by taking the lowest value of Y, then adding this value to both the *x_cord* and *z_cord* coordinates of the camera position vector.

Cumulus and stratocumulus clouds, requiring more random locations within the camera view compared to altostratus and cirrus, render a maximum of two clouds each and are positioned relative to the camera using offsets to relocate to new positions. The offsets are random values within the camera view used to scatter the clouds on the view. *cameraFront * moveDistance* moves the cloud along the direction the camera is facing, determined by *cameraFront*, by a distance specified by *moveDistance*. *cameraRight * xOffset* introduces a horizontal offset to the cloud's position based on the camera's rightward orientation (*cameraRight*), with the offset determined by *xOffset*. The *xOffset* value is multiplied by *cameraRight* to determine how far the cloud will move horizontally relative to the camera's orientation. *cameraUp * yOffset* adds a vertical offset based on the camera's upward orientation (*cameraUp*), with the offset determined by *yOffset*. The *yOffset* value is multiplied by *cameraUp* to determine how far the cloud will move vertically relative to the camera's orientation. This ensures that the clouds move in the direction the camera is facing and get various distances from the camera to simulate real-world clouds.

3.4 Lighting

In order to simulate lighting conditions on the clouds, we used Phong shading which consists of three different components: ambient, diffuse and specular [25]. In this case we only use ambient and diffuse lighting.

Ambient lighting represent the global illumination in a scene while diffuse lighting replicates the way light diffuses across uneven surfaces and gives the displayed scene more depth and realistic appearance.

Diffuse lighting is calculated as the dot product between the light-direction vector and the normal vector to a specific fragment. In order to replicate this effect on our 2D-cloud billboards, we have generated a normal vector for each fragment so that the fragments facing the light source will be brighter than fragments facing away from the light source. For a detailed description see Appendix B.7.

3.5 Experiments

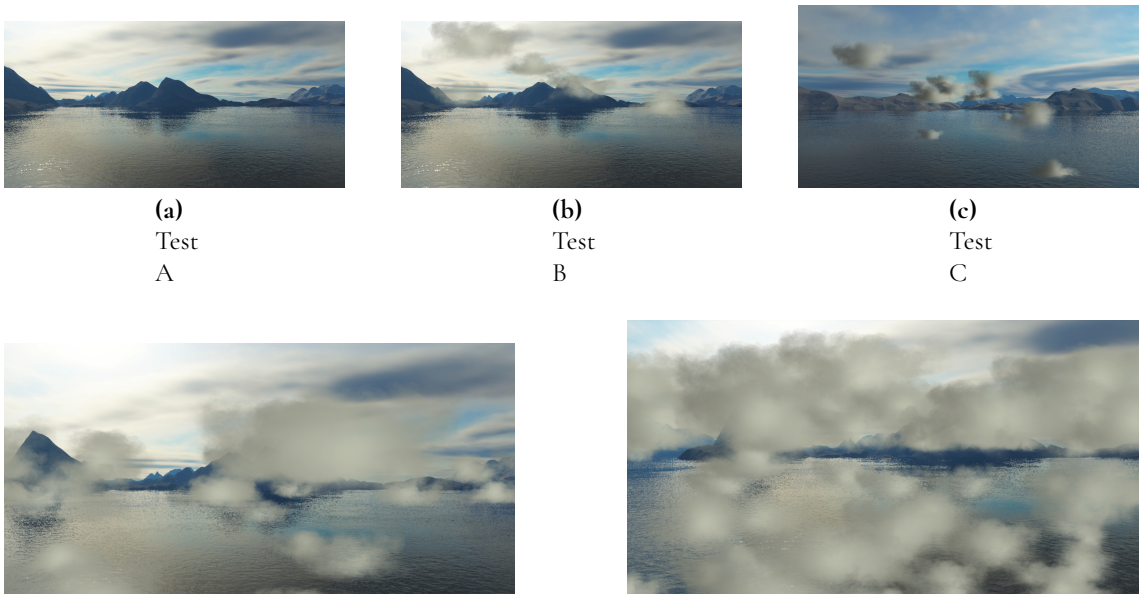
To assess our application's performance on the monitors used for Tactel's ARC engine, we've established specific performance tests. We will do a total of 90 tests with different parameters explained below on each monitor, gathering up to a total of 270 tests. We have created five different applications with different weather conditions and amount of clouds to be rendered. Each application will run five times for 20 seconds where data will be gathered for

further analysis. The collected data from the experiments will include CPU load and GPU frames per second (FPS), as these metrics are directly linked to the overall performance of the system. Please see Appendix B.8 for an explanation of how the data is gathered for the CPU and the GPU.

The data collection process involved manual execution. On our test computer, a data gathering script, detailed in Appendix B.4, is initiated once the application is configured and launched. The application is started manually by clicking on the application icon on the designated monitor under test. This testing procedure runs for a fixed duration of 20 seconds. Upon completion, the script automatically terminates and writes the collected data to designated files. It is important to note that the manual initiation of the application introduces potential for human error in the measurements

The five different applications have different amounts of clouds and would simulate different weather conditions. Please see Figure 3.5 and Appendix A.4 for an illustration of the test environment with different configurations.

- Test A: Zero clouds
- Test B: 10 clouds
- Test C: 30 clouds
- Test D: 75 clouds
- Test E: 200 clouds



(a) Test A

(b) Test B

(c) Test C

(d) Test D

(e) Test E

Figure 3.5: Tests A-E

In order to test the performance on the hardware we have conducted three tests.

3.5.1 Test case 1: Clustering

In the application there is a parameter for clustering the clouds. This clustering takes all the clouds and places them in a formation so that it forms one wide cloud in front of the camera. This is to test how the hardware performs with a high number of objects that need to be rendered and when all the clouds change angles on top of each other. The camera starts quite far away from the clouds and flies towards the clustered cloud to simulate a continuously moving airplane. When the camera has flown through the clustered formation of clouds, the frustum culling function will spread out the clouds to imitate a more real world simulation. This test case aims to demonstrate the efficiency of the hardware in swiftly adjusting from a clustered formation to a spread out formation, specifically after frustum culling has repositioned the clouds. The NEXT monitor will not undergo clustering testing due to its hardware limitations. It has been observed that the monitor's performance is insufficient, taking minutes to initiate and run the application. Furthermore, its clustering data is inaccurate, as it measures data during application boot-up, leading to erroneous results.

3.5.2 Test case 2: No Clustering

In this test case the clouds do not cluster as above when the application has started. Instead the clouds are more spread out than in the clustered test case. In the beginning the camera is facing all the clouds that the test provides, this in order to test how the hardware behaves if all the clouds appear in the camera view but more spread out. When the clouds are outside the frustum we will spread them out as explained before in the frustum culling function.

3.5.3 Test case 3: Optimization

A crucial technique in rendering with 2D objects is back-to-front rendering which has been discussed before. Since this requires the rendered objects to be sorted back to front, we sorted the array with cloud positions and copied the sorted data to the GPU on every frame. The optimization done in our code can be seen in Appendix B.9.

We realized that this could cause performance issues. Especially the need for copying the whole array to the GPU on every frame so we decided to experiment with only sorting and copying under some circumstances. Our optimized solution therefore only sorts and copies the data if the camera's position has substantially changed relative to the clouds. We can get this information from the frustum culling method since we already check the camera's view relative to the clouds. We therefore set a flag when the clouds end up outside of the view, if the flag is set then we sort the array and copy the data to GPU on the next frame.

This test is therefore a combination of test case 1 and test case 2 with an extension of the optimization described above. This test will test both clustering and no clustering with the optimization.

Chapter 4
Evaluation

This chapter will present the results from the experiments described in the previous chapter and the findings regarding how billboards suit clouds in an aerospace application and investigating hardware limitations thus giving data for answering **RQ1** and **RQ2**.

4.1 Results

4.1.1 Lighting and appearance

This video[1] shows how light affects the clouds with an artificial sun. The light is illustrated as a white quad that moves in a circle in the xy plane.

The Figures 4.1 and 4.2 below show different snapshots in different angles and different clustering configurations with the weather codes *D200 & D400* and snapshots of cirrus and altostratus:

[1]Demonstration of how light affects the clouds in the application: `https://youtu.be/Smr6hEMPlr8`

(a) D400 - clustered

(b) D400 - not clustered

(c) D400 - not clustered from above

(d) D200 - not clustered

Figure 4.1: Different cloud snapshots

(a) Altostratus

(b) Cirrus

Figure 4.2: Cirrus and altostratus in our application

4.1.2 Benchmarking - CPU

The data gathered from the tests for the CPU, which is the CPU load, can be seen in Tables 4.1, 4.2 & 4.3, where **Start** is the initial stage of the test, **Mid** is the middle part of the test (after 10 seconds) and **End** is the end of the test. After investigating the loads, which is in percentage (%) of the whole system, there does not seem to be a huge spike in CPU load on the tests except for the tests C and E which contain more clouds than the other tests. The tests C and E have an increased load of around 6-8% after the starting phase for the Astrova monitor, which could be problematic due to its potential impact on system performance and resource utilization, while the eX3 shows an increased load in test E with around 7% growth, which could also potentially impact system performance. For the NEXT monitor the CPU load seems relatively stable and does not seem to have an increasing or decreasing pattern. The other tests have some increase and decrease in CPU load but with less noticeable values.

The test B on the Astrova monitor has a high CPU load for optimised clustering and non-clustering at the start of the testing and stabilises in the middle to end of the testing. This phenomenon appears in test D also for the Astrova where all the different tests start with a high load and decreases during iterations whilst having a lower amount of clouds rendered.

Table 4.1: Clouds Analysis - Astrova

Test (Clouds)	Value (%)			
	Non-optimised Non-clustered	**Non-optimised Clustered**	**Optimised Non-clustered**	**Optimised Clustered**
A (0)	Start: 7.66 Mid: 8.0 End: 8.0	Start: 7.66 Mid: 8.0 End: 8.0	Start: 7.66 Mid: 8.0 End: 8.0	Start: 7.66 Mid: 8.0 End: 8.0
B (10)	Start: 9.6 Mid: 8.6 End: 8.6	Start: 9.32 Mid: 8.6 End: 8.6	Start: 12.94 Mid: 8.4 End: 8.6	Start: 12.36 Mid: 9.4 End: 8.6
C (30)	Start: 9.18 Mid: 11.2 End: 9.6	Start: 7.74 Mid: 10.2 End: 10.4	Start: 5.8 Mid: 13.8 End: 13.6	Start: 6.3 Mid: 13.4 End: 13.2
D (75)	Start: 13.84 Mid: 8.8 End: 8.8	Start: 15.2 Mid: 9.2 End: 9.2	Start: 14.86 Mid: 11.2 End: 11.0	Start: 13.62 Mid: 11.0 End: 10.2
E (200)	Start: 4.62 Mid: 11.6 End: 12.4	Start: 4.56 Mid: 10.8 End: 11.0	Start: 4.4 Mid: 10.8 End: 10.8	Start: 4.04 Mid: 10.8 End: 10.8

Table 4.2: Cloud Analysis - eX3

Test (Clouds)	Value (%)			
	Non-optimised Non-clustered	**Non-optimised Clustered**	**Optimised Non-clustered**	**Optimised Clustered**
A (0)	Start: 7.62 Mid: 6.8 End: 6.4	Start: 7.62 Mid: 6.8 End: 6.4	Start: 7.62 Mid: 6.8 End: 6.4	Start: 7.62 Mid: 6.8 End: 6.4
B (10)	Start: 8.78 Mid: 7.8 End: 7.8	Start: 9.56 Mid: 7.8 End: 8.0	Start: 8.92 Mid: 8.4 End: 8.34	Start: 8.02 Mid: 8.6 End: 8.2
C (30)	Start: 6.92 Mid: 10.0 End: 9.4	Start: 7.98 Mid: 9.8 End: 9.6	Start: 8.7 Mid: 10.0 End: 10.0	Start: 8.86 Mid: 10.2 End: 10.4
D (75)	Start: 11.22 Mid: 8.6 End: 8.2	Start: 11.54 Mid: 8.2 End: 8.4	Start: 9.94 Mid: 9.0 End: 8.6	Start: 10.84 Mid: 8.6 End: 8.2
E (200)	Start: 4.48 Mid: 11.2 End: 11.2	Start: 4.54 Mid: 12.2 End: 11.4	Start: 5.5 Mid: 12.0 End: 12.16	Start: 3.88 Mid: 9.4 End: 11.0

Table 4.3: Cloud Analysis - NEXT

Test (Clouds)	Value (%)			
	Non-optimised Non-clustered	**Non-optimised Clustered**	**Optimised Non-clustered**	**Optimised Clustered**
A (0)	Start: 2.0 Mid: 2.0 End: 2.0	Start: - Mid: - End: -	Start: 2.0 Mid: 2.0 End: 2.0	Start: - Mid: - End: -
B (10)	Start: 2.6 Mid: 3.0 End: 3.0	Start: - Mid: - End: -	Start: 2.4 Mid: 3.0 End: 2.8	Start: - Mid: - End: -
C (30)	Start: 3.0 Mid: 3.4 End: 3.2	Start: - Mid: - End: -	Start: 2.8 Mid: 3.0 End: 3.2	Start: - Mid: - End: -
D (75)	Start: 2.4 Mid: 2.4 End: 2.4	Start: - Mid: - End: -	Start: 3.0 Mid: 3.0 End: 2.6	Start: - Mid: - End: -
E (200)	Start: 2.2 Mid: 2.2 End: 2.0	Start: - Mid: - End: -	Start: 1.4 Mid: 2.2 End: 2.2	Start: - Mid: - End: -

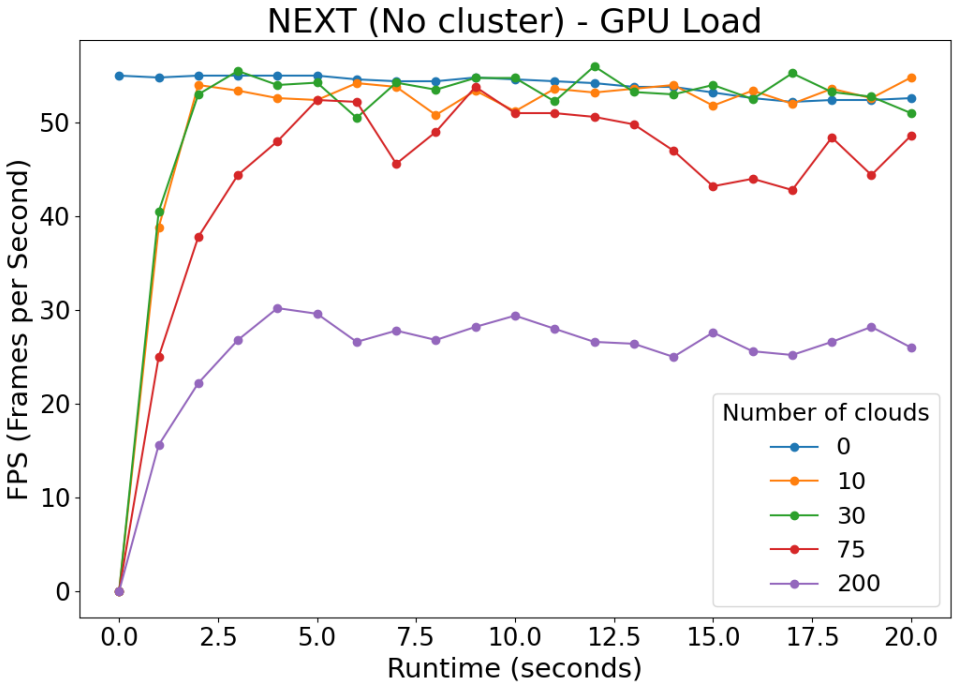
4.1.3 Benchmarking - GPU

The NEXT monitor in Figure 4.3a & 4.3b has quite similar FPS data in comparison with the no-clustering test case. Note that it was NOT possible to do testing with clustered properties hence the loss of two plots. With the optimization, the FPS in test E (with 200 clouds) seems to vary between 22 and 30 fps in a fluctuating pattern whilst test E with no optimization seems to keep it roughly constant. The other test data is very similar with the only notable difference in test D (with 75 clouds) having the same fluctuating pattern with optimization indicating something unstable in the performance.

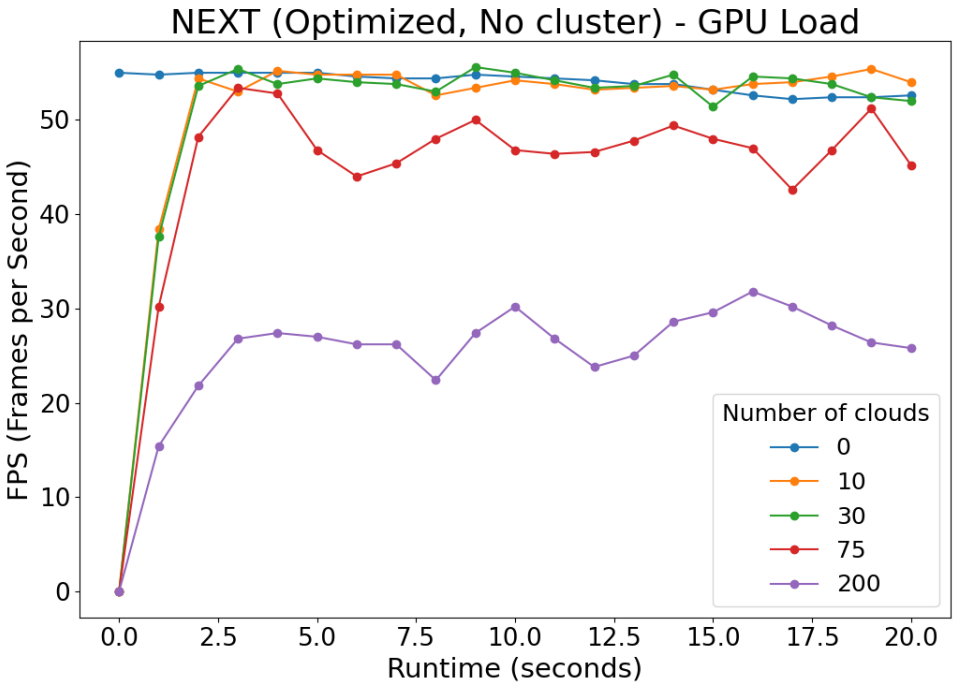
For the eX3 monitor the patterns for the four plots seem to resemble each other. The tests D (75) and E (200) start relatively low during runtime and stabilise after roughly five seconds. Performance wise, the eX3 seems to handle higher cloud numbers, both with optimization and clustered, better as the tests seem to stay around 50 FPS after some runtime. In Figure 4.4c & 4.4d there is a big difference in results on tests D (75) and E (200) where the no-optimization achieves a stable FPS faster and maintains it, while the optimization achieves higher FPS faster but in test E it fluctuates around 35 and 47 FPS during runtime like the pattern explained above with the NEXT monitor. All the tests except for test E seem to stabilise quite fast and maintain a stable 50 FPS.

The Astrova monitor has, despite its newer software and hardware, some very interesting results. For the tests with lower cloud numbers, the monitor performs very well with 60 FPS and reaches this state fairly quickly. For test D (75) it takes roughly five seconds to achieve 60 FPS and seems to stay at around 30 to 50 FPS while stabilising towards 60 FPS. In this test, the difference in reaching 60 FPS is fairly minimal with optimization and clustering. The interesting part is test E (200) on the four different test cases. With this amount of clouds it takes around eight seconds to achieve the stable state for the FPS. However, this stable state

for the test cases is around 40 FPS, which is lower than the eX3 monitor. One observation with the Astrova monitor is that the non optimized tests seem to have this fluctuating pattern instead of the optimized tests which are contrariwise for the NEXT and eX3 monitors.

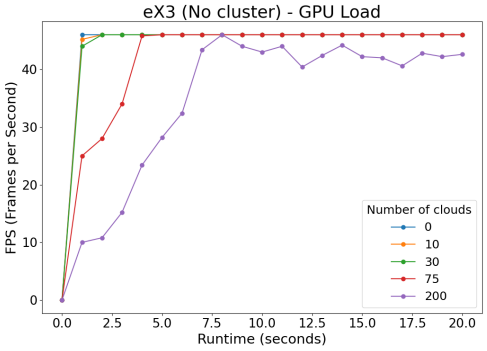


(a) NEXT - no optimization, no cluster

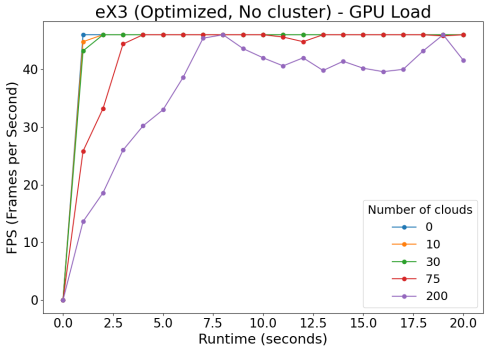


(b) NEXT - optimization, no cluster

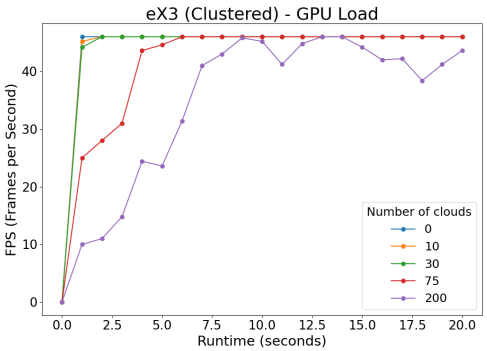
Figure 4.3: GPU data (FPS) for the NEXT monitor



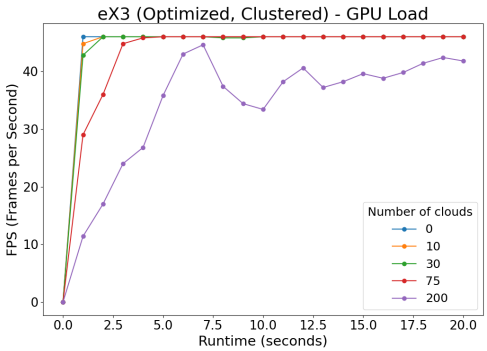
(a) eX3 - no optimization, no cluster



(b) eX3 - optimization, no cluster

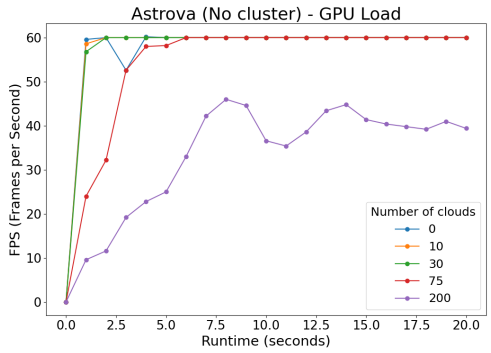


(c) eX3 - no optimization, cluster

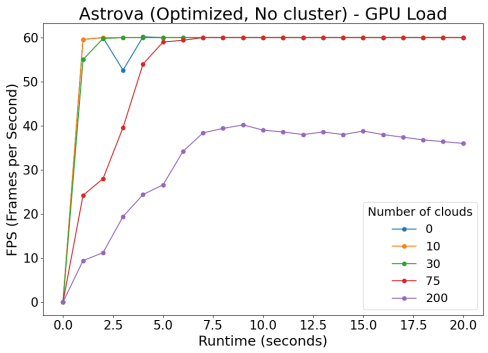


(d) eX3 - optimization, cluster

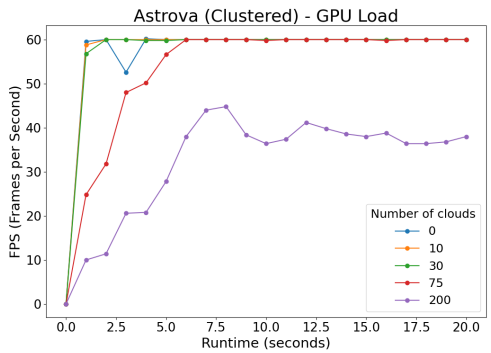
Figure 4.4: GPU data (FPS) for the eX3 monitor



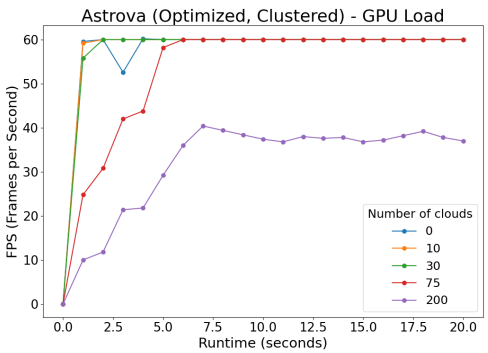
(a) Astrova - no optimization, no cluster



(b) Astrova - optimization, no cluster



(c) Astrova - no optimization, cluster



(d) Astrova - optimization, cluster

Figure 4.5: GPU data (FPS) for the Astrova monitor

Chapter 5

Discussion

In this chapter, the results from the experiments are analysed, with a focus on answering this thesis' research questions. The outcome of this thesis is discussed, together with limitations provided by this solution, and some future work in which our thesis can be used as a foundation.

5.1 Cloud rendering with billboards

The results gave some important findings regarding the cloud rendering of the different cloud structures with the use of billboards. This is analysed in the following sections.

5.1.1 Cumulus & Stratocumulus

Cumulus and stratocumulus clouds often have well-defined shapes that can be effectively represented using billboards. These cloud types frequently exhibit repetitive patterns across the sky, making them suitable candidates for billboarding. By using a small set of billboard images and cleverly arranging them in a repeating pattern, it is possible to simulate the appearance of vast expanses of cumulus or stratocumulus clouds without needing a large library of unique textures. This of course could lead to the simulation being less realistic due to the repeating patterns of clouds. This static nature makes them ideal for the billboard technique, as the images used to represent them do not need frequent updates to match evolving cloud formations. As a result, the lack of depth perception inherent in billboards is less noticeable when representing these lower-altitude cloud types, making them appear more realistic in the rendered scene.

5.1.2 Altostratus & Cirrus

These clouds are known for being broad in the sky. This means that in the simulation of cloud rendering of these two cloud types they need to be very broad in size in the engine to create a more realistic visualisation for the passengers. This however creates bigger problems for the billboard technique. The billboard technique as mentioned before is changing its angle from the center point of the object to be rendered towards the camera view in order to simulate a 3D object with a 2D plane. Objects this size create issues, by being 2D images that always face the camera and cannot accurately convey the three-dimensional nature of these clouds. Using billboards for such clouds would result in unrealistic flat representations that lack depth and perspective. The cloud would face the camera and simply look like a small cloud which is not that dense.

Instead of the billboard technique, rendering cirrus and altostratus clouds in computer graphics typically involves more advanced methods such as volumetric rendering or particle systems.

5.2 Benchmarking Results

5.2.1 CPU Load investigation

After investigating the results in the Evaluation section there does not seem to be any direct impact on the CPU load except for when there are higher cloud numbers to be rendered.

A general explanation of why the test E, which has the highest cloud amount of 200, has a higher impact on CPU load than the other tests is due to that the GPU and CPU focus on different tasks. The CPU in computer graphics focuses on where the objects to be rendered are supposed to be located and how many of them there are, whilst the GPU focuses on what the objects should look like [35, 30]. In computer graphics it is quite unusual that test C (30 clouds) on the Astrova monitor has gained that much load on the CPU as there is a low amount of objects during this test.

As seen in the tables, a vast majority of the tests show a decrease in load after the application has run for a while. The increased CPU load at the start could be caused by the application's startup when all the clouds are generated with random locations within the camera's view. This means that, for example, in test D, which has 75 clouds, all 75 clouds must be assigned a position and undergo our sorting process to prevent graphical issues with the billboards. During runtime, the clouds change positions when they are outside of the frustum. In theory, this should not affect the load as much as the scenario in the initial phase of the application, which explains the decrease in CPU load. For test E, there is simply too many clouds that relocate, causing a higher impact on the load.

5.2.2 Initial drop in performance

We can see for almost all the results that there is a drop in FPS in the initial phase of runtime but then the FPS stabilizes and does not fluctuate as much after some time. This is a predicted result which depends on the way we first place the clouds and how they are later spread once the camera passes through them. At first they are usually tightly packed and cover large

portions of the screen but after the camera passes through they get spread out along the whole view which potentially increases the FPS.

One reason performance drops when we have tightly packed billboards covering a small area and a large portion of the screen simultaneously is overdraw. Overdraw occurs when multiple objects are rendered onto the same pixel regions of the screen [27]. When the cloud billboards are packed tightly and cover large portions of the screen, each pixel area may be processed multiple times as the GPU renders each overlapping cloud. Additionally, since blending is enabled and all our clouds are translucent, many calculations are required for each pixel.

Another reason for performance drops could be texture sampling. When the fragment shader needs to sample many billboards that cover large portions of the screen, this can cause calculation overhead. A possible solution would be to test using Level of Detail (LOD), which involves using lower resolutions for billboards that are further away from the camera.

Another cause could be resolution on the monitors. The Panasonic eX3 and NEXT monitor run in *1080p (1920 x 1080 pixels)* [4, 5] while the Astrova monitor uses *4K (3840 x 2160 pixels)* [3]. Resolution refers to the number of pixels displayed on the screen. Higher resolutions, such as 4K, require more processing power to render due to a higher number of pixels in comparison to a 1080p screen, resulting in increased GPU usage. The GPU has to use more resources to process and display the additional pixels. This is because higher pixel monitors, like the Astrova, require more processing power from the GPU. If the GPU is not well-suited for the monitor's resolution, the GPU load could significantly increase.

5.2.3 Comparison of Clustering vs. Non-Clustering

The results do not show a clear difference between different clustering configurations in terms of improving performance across all three screens. Only when the clouds are very tightly packed in the start do we see a drop in performance that we discussed above. In our implementation, both the clustered and non-clustered configurations tend to be tightly packed at the start of runtime and then depending on the clustering configuration get spread out either with larger distances in between them for non-clustered, or with more tight distances for clustered clouds.

In some situations however, especially for Astrova, clustered clouds perform slightly better than non-clustered counterparts. These are not frequent enough though to draw the conclusion that clustering offers any benefits. This implies that the clustering configuration used does not significantly affect performance as long as the clouds are spread out along the whole view range and are not all packed in a small area.

5.2.4 Evaluation of Optimization Techniques

The performance is slightly enhanced by optimization on all displays and with all configurations. This emphasizes how crucial the optimization technique is to the rendering processes, since it can greatly improve FPS and CPU use. The results clearly support the creation and application of efficient algorithms for rendering tasks in order to improve performance and manage resources more effectively.

5.3 Limitations

Due to time constraints of this thesis there have been some limitations. We wanted to create more tests with additional clouds and other optimizations which we did not have time to implement. Testing on something other than the monitors that we used would also be interesting to see how good or bad the monitors perform in comparison to modern PCs with e.g. RTX4000 series graphic cards and newer processors.

Extending the test scenarios to include more clouds and adding other optimizations could improve the results' readability. Furthermore, testing on other displays would provide insights about performance and applicability in real-world scenarios. With a more comprehensive grasp of the system's capabilities, the results of this testing strategy should be easier to comprehend.

5.4 Integration of clouds in the ARC engine

As stated before the monitors run at around 47 FPS when using the ARC engine on the different monitors. However, the Android application that renders the clouds in some cases stabilizes at a frame rate that is lower than this for cases such as 75 and 200 clouds. This indicates that for integration of clouds in the ARC engine the need for better optimizations and finding a good match on how many clouds to render based on performance on the system is necessary. The goal is to not lower the FPS much further than 47 FPS when ARC is running on the systems. However, the performance on the systems seems good enough to still use cloud rendering with billboards but any further considerations as explained above must be taken into account when integrating into the ARC engine.

5.5 Future Work

For future implementations of our thesis project regarding performance and user experience there are some topics to discuss. The use of volumetric or particle systems is something that could be tested further in comparison to billboarding. Particle and volumetric systems are a better choice than billboarding in computer graphics due to their realism and flexibility. They simulate complex phenomena more accurately, preserve volume and interact with lighting and shading effectively. However, they may demand more computational resources, advancements in hardware and software optimizations make them feasible for real-time rendering, enhancing user experience in graphical applications. The questions regarding performance and user experience which is the better for the customers in this case is raised.

Further the use of stochastic transparency which we did not manage to explore enough could possibly improve the results of the thesis or give it better data. Stochastic transparency eliminates the need for sorting transparent objects in complicated settings making the application less computationally demanding [9]. It lessens the requirement for exact sorting by adding randomization to transparency computations, possibly allowing for speedier rendering without compromising visual quality if the noise is handled correctly. The difference in performance and realism between traditional sorting-based blending algorithms and stochastic transparency is therefore worth exploring for this application.

If the future hardware increases in processing power and acquire more powerful GPUs for the systems, the possibility to utilize more advanced rendering techniques for even more realistic imaging is possible. This is something that could be researched a few years later if the hardware has been improved and if the future systems are more capable of utilizing other rendering techniques.

Chapter 6
Conclusion

This thesis strives to help the case company in evolving their product to gain a more pleasant user experience for the customers by testing if the current systems are capable of cloud rendering with the hardware they have today. In this thesis the Billboard method, which orientates the object to always face the camera whichever way the camera is looking, has been used for creating realistic representations of clouds.

The investigation process for the described problem was to use performance measuring to see if the current generation of in-flight entertainment systems from Panasonic is ready for heavy rendering of weather phenomena which are present in these clouds. The testing has been done with different amounts of clouds with some optimizations and clustering to see how the system can handle different settings for the cloud rendering.

In order to answer **RQ1**, different optimizations had to be implemented. The two notable optimizations were frustum culling and instancing. Frustum culling disables the rendering of the objects to save resources and relocates them if outside the frustum, while instancing ensures less traffic between the CPU and the GPU by issuing one draw call over all the objects. An implementation of the billboarding technique can simulate 3D depth by rendering 2D sprites from various angles giving a realistic representation from various spatial perspectives. Also, the optimization with back-to-front sorting only when the camera's position has substantially changed relative to the clouds proved to cause impact on the performance. Given the results and the optimizations tested, the current hardware of in-flight entertainment systems seem ready for real-time rendering of clouds. But as discussed before the performance on the ARC engine with clouds could lower the overall performance on the systems.

In order to answer **RQ2**, an investigation of how the ARC engine fetches the data was required. By fetching Geolocation data from Geonames and then weather data from Foreca, we get both the weather type and its temperature. This approach was tried out until we had to change to a stand-alone Android project, blocking the opportunity to try this out with real-time data. However, our provided solution still uses the weather codes fetched from Foreca to simulate real-time data and renders clouds based on the weather codes *D000, D100, D200, D300* & *D400*. The realistic cloud design comes from the implemented shader

which uses Phong shading and alpha mixing. By scattering the clouds on the camera view inside the frustum the user sees a realistic representation of clouds depending on the weather code where the altostratus and cirrus clouds reside higher up in the sky to simulate realistic properties. By sorting the clouds with back-to-front rendering, the target cloud will not disrupt the rendering by the clouds behind which gives a realistic appearance of how clouds work out even if they are at approximately the same location.

Although the proposed solution and results in this thesis might not fully solve the problems at the case company, the research questions were answered, and the implementation used in this thesis can be used in a real-world scenario. The results of this thesis are a good starting point of how to optimize and continue the research of cloud rendering for in-flight entertainment systems.

References

[1] Tactel ab. https://tactel.se/sv/. Accessed on 2024-03-20.

[2] adonis one. Top 6 advantages of in-flight entertainment systems adonisone. https://www.linkedin.com/pulse/top-6-advantages-in-flight-entertainment-systems-adonisone-adonis-one/, 2023. Accessed on 2024-03-20.

[3] Panasonic Avionics. Astrova. https://www.panasonic.aero/our-offerings/systems/astrova/, 2024. Accessed on 2024-04-15.

[4] Panasonic Avionics. Next series. https://www.panasonic.aero/our-offerings/systems/next-series/, 2024. Accessed on 2024-04-15.

[5] Panasonic Avionics. X series. https://www.panasonic.aero/our-offerings/systems/x-series/, 2024. Accessed on 2024-04-15.

[6] Harriet Baskas. Customer satisfaction with airlines is rising — as long as the in-flight entertainment is good. https://www.nbcnews.com/business/travel/customer-satisfaction-airlines-rising-long-inflight-entertainment-good-n8786
2018. Accessed on 2024-03-20.

[7] Lars Behrendt and Leif Kobbelt. Realistic and real-time rendering of landscapes. https://graphics.uni-konstanz.de/publikationen/Behrendt2005Realisticrealtime/Behrendt2005Realisticrealtime.pdf, 2005. Accessed on 2023-02-02.

[8] Android Developers. Android debug bridge (adb). https://developer.android.com/tools/adb, 2024. Accessed on 2024-04-15.

[9] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic transparency. In *I3D '10: Proceedings of the 2010 symposium on Interactive 3D graphics and games*, pages 157–164, New York, NY, USA, 2010.

[10] Stream HPC. Vivante gpu (freescale i.mx6). https://streamhpc.com/knowledge/sdks/vivante-gpu/, 2023. Accessed on 2024-04-12.

[11] Six Jonathan. Frustum culling. https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling, 2021. Accessed on 2024-02-24.

[12] LaVision. Mie, rayleigh, raman imaging techniques. https://www.lavision.de/en/techniques/mie-rayleigh-raman/, Year. Accessed on 2024-02-04.

[13] LearnOpenGL. Blending. https://learnopengl.com/Advanced-OpenGL/Blending, 2015. Accessed on 2024-02-23.

[14] LearnOpenGL. Instancing. https://learnopengl.com/Advanced-OpenGL/Instancing, 2015. Accessed on 2024-03-11.

[15] Lenovo. Lenovo sprite glossary. https://www.lenovo.com/us/en/glossary/sprite/?orgRef=https%253A%252F%252Fwww.google.com%252F, Year. Accessed on 2024-02-05.

[16] Patrick T Magee. Chapter 15 - physiological monitoring: Gasses. In Andrew J Davey and Ali Diba, editors, *Ward's Anaesthetic Equipment (Sixth Edition)*, pages 337–350. Elsevier, Edinburgh, sixth edition edition, 2012. Accessed on 2024-02-04.

[17] Lukas Mattsson. Simulating optical depth for participating media using ray tracing. https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9110658&fileOId=9110660, 2022. Accessed on 2024-02-01.

[18] Filip Nilsson. 3d cloud visualization in real-time. http://www.diva-portal.org/smash/get/diva2:1673270/FULLTEXT01.pdf, 2022. Accessed on 2024-01-26.

[19] Notebookcheck. Qualcomm adreno 530. https://www.notebookcheck.net/Qualcomm-Adreno-530.156189.0.html, 2024. Accessed on 2024-04-15.

[20] National Oceanic and Atmospheric Administration (NOAA). The color of clouds. https://www.noaa.gov/jetstream/clouds/color-of-clouds, 2023. Accessed on 2024-01-29.

[21] Rikard Olajos. Real-time rendering of volumetric clouds. https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8893256&fileOId=8893258, 2016. Accessed on 2024-02-01.

[22] Sean O'Neil. *Chapter 16: Accurate Atmospheric Scattering*, chapter 16. NVIDIA Corporation, 2005. Accessed on 2024-02-04.

[23] DPSTUDIO SP. Z O.O. Volumetric clouds. https://blendermarket.com/products/volumetric-clouds, 2023. Accessed on 2024-06-09.

[24] OpenGL Tutorial. Billboards. https://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/billboards/, 2024. Accessed on 2024-02-05.

[25] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, jun 1975.

[26] Planetside Software. Dot Product. https://planetside.co.uk/wiki/index.php?title=Dot_Product, 2022. Accessed on 2024-03-18.

[27] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.*, 26(3):89–es, jul 2007.

[28] NXP Semiconductors. i.mx 6quad processors - high-performance, 3d graphics, hd video, arm® cortex®-a9 core. https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-6-processors/i-mx-6quad-processors-high-performance-3d-graphics-hd-video-arm-cortex-a9-co i.MX6Q, 2024. Accessed on 2024-04-12.

[29] Passmark Software. Qualcomm technologies, inc apq8096pro. https://www.cpubenchmark.net/cpu.php?cpu=Qualcomm+Technologies%2C+Inc+APQ8096pro&id=4970, 2024. Accessed on 2024-04-15.

[30] SoftwareG. Game settings that affect cpu. https://softwareg.com.au/blogs/computer-hardware/game-settings-that-affect-cpu, 2024. Accessed on 2024-05-04.

[31] Tactel. Exploring the world below from the sky above. https://tactel.se/en/cases/exploring-the-world-below-from-the-sky-above/, 2024. Accessed on 2024-01-26.

[32] Qualcomm Technologies. Apq8096sg. https://www.qualcomm.com/products/technology/processors/application-processors/apq8096sg#Overview, 2024. Accessed on 2024-04-15.

[33] Qualcomm Technologies. Snapdragon 820 mobile platform. https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-820-mobile-platform, 2024. Accessed on 2024-04-15.

[34] Eric Haines Tomas Akenine-Möller. Billboarding. https://www.flipcode.com/archives/Billboarding-Excerpt_From_iReal-Time_Renderingi_2E.shtml, 2008. Accessed on 2024-02-05.

[35] Reddit user Cutter9792. Can graphics affect cpu performance? https://www.reddit.com/r/buildapc/comments/15iwlqy/can_graphics_affect_cpu_performance/?rdt=58661, 2023. Accessed on 2024-05-04.

[36] Lode Vandevenne. Raycasting. https://lodev.org/cgtutor/raycasting.html, 2020. Accessed on 2024-01-30.

[37] Niniane Wang. Let there be clouds: Fast, realistic cloud rendering in *Microsoft Flight Simulator 2004: A Century of Flight*, 2004. Accessed on 2024-01-26.

[38] Wikibooks. Cg programming/unity/billboards, 2022. Accessed ON 2024-02-05.

[39] Wikipedia. Altostratus. https://en.wikipedia.org/wiki/Altostratus_cloud, 2023. Accessed on 2024-01-26.

[40] Wikipedia. Cirrus. https://en.wikipedia.org/wiki/Cirrus_cloud, 2023. Accessed on 2024-01-29.

[41] Wikipedia. Cumulus. https://en.wikipedia.org/wiki/Cumulus_cloud, 2024. Accessed on 2024-01-26.

[42] Wikipedia. Stratocumulus. https://en.wikipedia.org/wiki/Stratocumulus_cloud, 2024. Accessed on 2024-01-26.

Appendices

Appendix A
Images

A.1 ARC

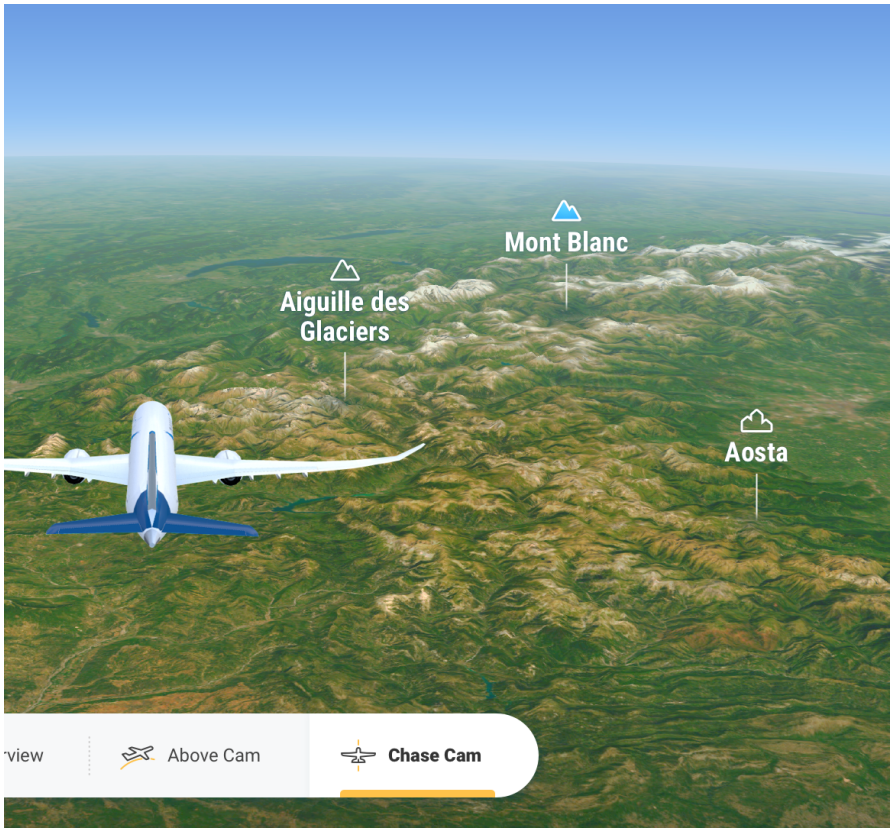


Figure A.1: Chase cam of the ARC in-flight entertainment system. Source: Tactel

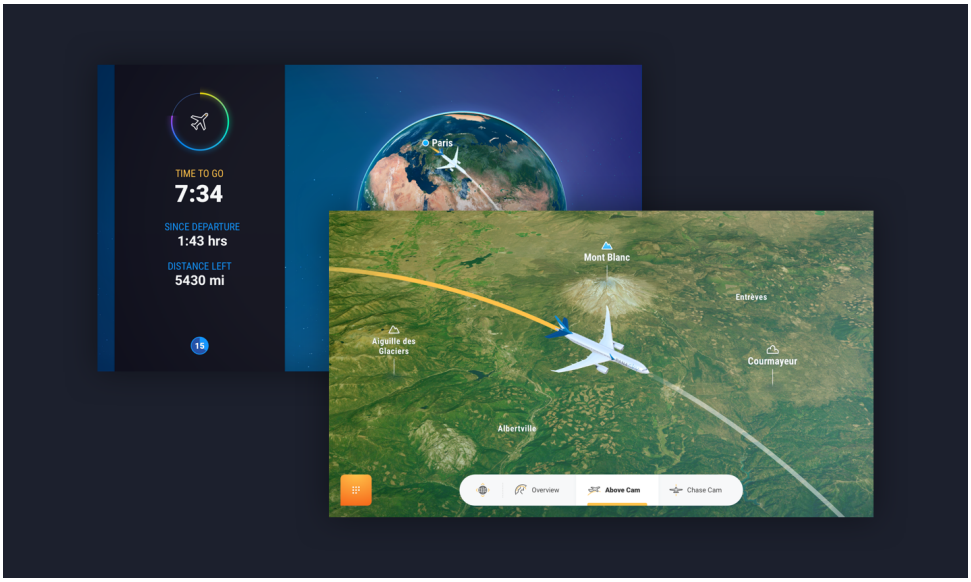


Figure A.2: Above cam of the ARC in-flight entertainment system. Source: Tactel

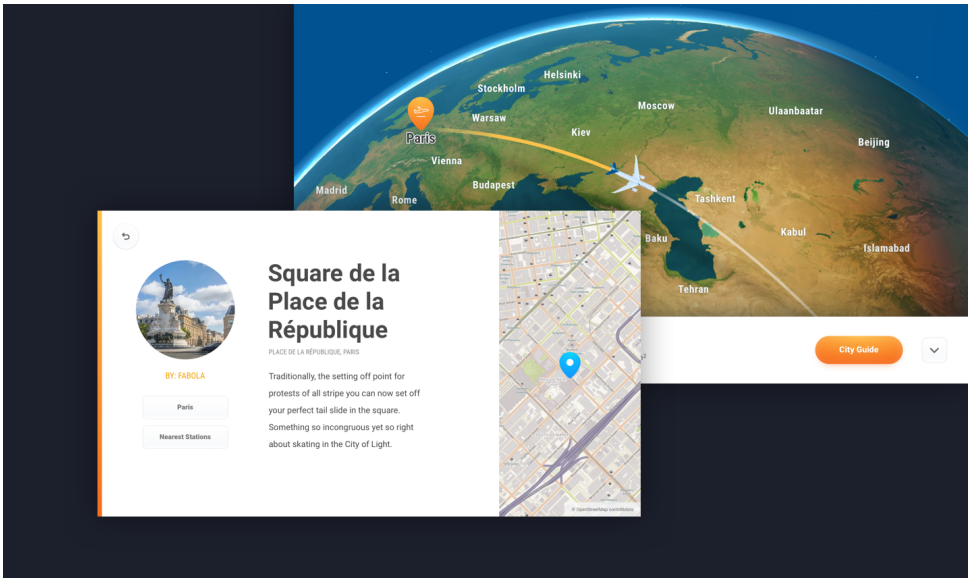


Figure A.3: Local guide for tourism in the ARC system

A.2 Application screenshot

Figure A.4: Demonstration of the Android application with the weather code D400 showing cumulus and stratocumulus clouds

Appendix B
Code

B.1 Frustum Culling

```
const float maxCumulusHeight = 5.0f;
const float maxAltostratusHeight = 35.0f;
const float maxCirrusHeight = 55.0f;
const float maxStratocumulusHeight = 7.0f;

glm::vec3 cameraRight = glm::normalize(glm::cross(
                        cameraFront, cameraUp));

for(int i = 0; i < numClouds; i++) {
    glm::vec3 toCloud = cloudPositions[i].position - cameraPos;
    float dotProduct = glm::dot(toCloud, cameraFront);

    if (dotProduct < 0.0f && glm::length(toCloud) > 2.0f) {
        float moveDistance = (rand() % 35) + 7.5f;

        float xOffset, yOffset = ((float)rand() /
                    RAND_MAX * 2.0f - 1.0f) * moveDistance;

        float x_cord = cameraFront.x * moveDistance;
        float y_cord = cloudPositions[i].position.y + moveDistance;
        float z_cord = cameraFront.z * moveDistance;

        int cloudType = cloudPositions[i].type;
        float altostratus_new_y = 0.0f;
        float cirrus_new_y = 0.0f;
        switch (cloudType) {
            case 0: // Cumulus
                cloudPositions[i].position = cameraPos +
                    cameraFront * moveDistance + cameraRight *
                    xOffset + cameraUp * yOffset;
```

```
                cloudPositions[i].position.y = std::min(
                    cloudPositions[i].position.y, maxCumulusHeight
                );
                break;
            case 1: // Altostratus
                altostratus_new_y = std::min(
                    cloudPositions[i].position.y,
                    maxAltostratusHeight
                );
                cloudPositions[i].position = cameraPos +
                    glm::vec3(x_cord, altostratus_new_y, z_cord);
                break;
            ...
```

Listing B.1: Frustum Culling implementation

B.2 Clustered weather re-locator

```
void adjust_cloud_positions_for_clustered_weather(Data*
    cloudPositions, int numClouds, float clusteringFactor) {
    for (int i = 0; i < numClouds; ++i) {
        for (int j = i + 1; j < numClouds; ++j) {
            // Check if both clouds are Stratocumulus or Cumulus
            if ((cloudPositions[i].type == 0
                || cloudPositions[i].type == 3)
                && (cloudPositions[j].type == 0
                || cloudPositions[j].type == 3) {

                // Calculate distance between clouds
                float distance = glm::distance(
                    cloudPositions[i].position,
                    cloudPositions[j].position);

                // If distance < threshold, adjust position
                if (distance < clusteringFactor) {
                    // Move clouds closer together
                    glm::vec3 direction = glm::normalize(
                        cloudPositions[j].position -
                        cloudPositions[i].position);
                    cloudPositions[i].position -= direction *
                        (clusteringFactor - distance) * 0.5f;
                    cloudPositions[j].position += direction *
                        (clusteringFactor - distance) * 0.5f;
                }
            }
        }
    }
}
```

Listing B.2: Change distance if clustered weather

B.3 Move nearby clouds based on cloud type

```
void get_nearby_cloud_positions_and_move(Data* cloudPositions, int
    numClouds, const char* weather_type) {
    if (strcmp(weather_type, "d300") == 0
        || strcmp(weather_type, "d400") == 0) {
        return;
    }

    float minDistance = 10.0f;

    for(int i = 0; i < numClouds; i++) {
        glm::vec3 currentCloud = cloudPositions[i].position;

        for(int j = 0; j < numClouds; j++) {
            if(i != j) {
                if ((cloudPositions[i].type == 1 &&
                    cloudPositions[j].type == 1)
                    || (cloudPositions[i].type == 2 &&
                    cloudPositions[j].type == 2)) {
                    // Altostratus and Cirrus clouds not
                    // too close to each other
                        cloudPositions[i].position.z -= 20.0f;
                } else {
                    glm::vec3 otherCloud =
                        cloudPositions[j].position;

                    // Calculate distance between
                    // current cloud and other cloud
                    float distance =
                        glm::distance(currentCloud, otherCloud);

                    // Check if distance is less than minimum
                    // allowed distance and change the position
                    // of the current cloud
                    if(distance < minDistance) {
                        glm::vec3 direction =
                            glm::normalize(
                            currentCloud - otherCloud);

                        cloudPositions[i].position +=
                            direction * (minDistance - distance)
                            * 0.5f;
                    }
                }
            }
        }
    }
}
```

Listing B.3: Increase the distance from the clouds

B.4 Data Collection script

```
from concurrent.futures import ThreadPoolExecutor
import subprocess
import time
```

```
import csv
import os
import math

# 20 second test, 5 tests per D###, 2 experiments per type (clustered, non-clustered)

"""
d000 - 0 clouds
d200 - 10 clouds
d400 - 75 clouds
d500 - 30 clouds // Our own version of a cloud type
d600 - 200 clouds // Our own version of a cloud type
"""

def run_subprocess(command):
    data = []

    NEXT = False

    if command == "":
        start_time = time.time()
        while time.time() - start_time < 20:
            time.sleep(20)
            result = subprocess.run(
                'adb shell "cat /sdcard/Download/fps.txt"'
            , shell=True, capture_output=True, text=True)
            output = result.stdout
            data.append(output.split("\n")[:len(output.split("\n")) - 1])

    else:
        start_time = time.time()
        result = subprocess.run(command, shell=True,
            capture_output=True, text=True
        )
        output = result.stdout.split("\n")
        tmp = []
        for line in output:
            if line.split() == []:
                continue

            if NEXT:
                cpu_load = line.split()[2]
            else:
                cpu_load = line.split()[8]

            if NEXT:
                cpu_load_value = float(cpu_load.rstrip('%'))
                formatted_output = "{:.1f}".format(cpu_load_value)
                tmp.append(formatted_output)
            else:
                tmp.append(cpu_load)
        data = tmp

    return data
```

```
58
59 # Configuration START
60 testing_cloud = {
61     'd000': False,
62     'd200': False,
63     'd400': False,
64     'd500': False,
65     'd600': False
66 }
67
68 testing_monitor = {
69     'NEXT': False,
70     'eX3': False,
71     'Astrova': False
72 }
73
74 clustred = False
75 optimised = False
76
77 testing_cloud_name = None
78 for var_name, value in testing_cloud.items():
79     if value:
80         testing_cloud_name = var_name
81         break
82
83 testing_monitor_name = None
84 for var_name, value in testing_monitor.items():
85     if value:
86         testing_monitor_name = var_name
87         break
88
89 if not testing_monitor_name:
90     print("No monitor selected")
91     exit()
92
93 if not testing_cloud_name:
94     print("No cloud type selected")
95     exit()
96
97 cpu_file_name = f'{testing_monitor_name}_{testing_cloud_name}_CPU.
    csv'
98
99 fps_file_name = f'{testing_monitor_name}_{testing_cloud_name}_GPU.
    csv'
100
101 if clustred:
102     if optimised:
103         fps_path = f"DATA_MEASUREMENT/optimised/clustered/{
    testing_cloud_name}/" + fps_file_name
104
105         cpu_path = f"DATA_MEASUREMENT/optimised/clustered/{
    testing_cloud_name}/" + cpu_file_name
106     else:
107         fps_path = f"DATA_MEASUREMENT/clustered/{testing_cloud_name
    }/" + fps_file_name
108
```

```
109         cpu_path = f"DATA_MEASUREMENT/clustered/{testing_cloud_name
    }/" + cpu_file_name
110 else:
111     if optimised:
112         fps_path = f"DATA_MEASUREMENT/optimised/non-clustered/{
    testing_cloud_name}/" + fps_file_name
113 
114         cpu_path = f"DATA_MEASUREMENT/optimised/non-clustered/{
    testing_cloud_name}/" + cpu_file_name
115     else:
116         fps_path = f"DATA_MEASUREMENT/non-clustered/{
    testing_cloud_name}/" + fps_file_name
117 
118         cpu_path = f"DATA_MEASUREMENT/non-clustered/{
    testing_cloud_name}/" + cpu_file_name
119 # Configuration END
120 
121 # Check if the file already exists
122 file_exists = os.path.exists(cpu_path) and os.path.exists(fps_path)
123 
124 # Open the file in append mode if it exists, otherwise open in
    write mode
125 mode = 'a' if file_exists else 'w'
126 
127 # Run the commands in parallel
128 commands = [
129     f"",
130     f"adb shell top -d 1 -n 20 | grep com.example.cl"
131 ]
132 
133 with ThreadPoolExecutor(max_workers=len(commands)) as executor:
134     outputs = executor.map(run_subprocess, commands)
135 
136 # Convert the map object to a list to access the outputs
137 outputs = list(outputs)
138 FPS = outputs[0][0]
139 CPU = outputs[1]
140 
141 # Write FPS data to CSV
142 with open(fps_path, mode, newline='') as f:
143     writer = csv.writer(f)
144     if not file_exists:
145         writer.writerow(['Time', 'FPS'])
146 
147     for parameters in FPS:
148         time, fps = parameters.split(',')
149         writer.writerow([time, fps])
150 
151     writer.writerow("-")
152 
153 # Write CPU data to CSV
154 with open(cpu_path, mode, newline='') as f:
155     writer = csv.writer(f)
156     if not file_exists:
157         writer.writerow(['CPU Load'])
158     for cpu_load in CPU:
```

```
        writer.writerow([cpu_load])
    writer.writerow("-")
```

B.5 Plot GPU data script

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

PATH_CLUS = 'DATA_MEASUREMENT/clustered'
PATH_NO_CLUS = 'DATA_MEASUREMENT/non-clustered'
PATH_OPT_CLUS = 'DATA_MEASUREMENT/optimised/clustered'
PATH_OPT_NO_CLUS = 'DATA_MEASUREMENT/optimised/non-clustered'

weather_codes = [("000", 0), ("200", 10), ("500", 30),
                 ("400", 75), ("600", 200)] # With their
    corresponding test amount of clouds

monitor = "NEXT" # NEXT, eX3 or Astrova

file_path_comp = [
    f'{PATH_OPT_NO_CLUS}/d{code}/{monitor}_d{code}_GPU.csv'
        for code, _ in weather_codes
]

def prepare_data(file_path):
    df = pd.read_csv(file_path)

    gpu_load = [x for x in df['FPS']]

    # Prepare individual measurements
    plot_data = []
    temp_plot_data = []
    for measurement in gpu_load:
        if np.isnan(measurement):
            plot_data.append(temp_plot_data)
            temp_plot_data = []
        else:
            temp_plot_data.append(measurement)

    # Get biggest measurement in length
    mean_data = []
    biggest_measurement = 0
    for data in plot_data:
        if len(data) > biggest_measurement:
            biggest_measurement = len(data)

    # Extend data to biggest length if needed
    for data in plot_data:
        if len(data) < biggest_measurement:
            data = np.append(data, [np.nan]*(
                biggest_measurement-len(data))
            )
        mean_data.append(data[:21])

```

```
    # Mean data calculation
    mean_load = []
    for i in range(21):
        mean = 0
        count = 0
        for data in mean_data:
            if not np.isnan(data[i]):
                mean += data[i]
                count += 1
        mean_load.append(mean/count)

    return mean_load


def plot_cpu_data(data):
    plt.figure(figsize=(12, 8))

    measure_counter = len(data)

    for d in (data):
        plt.plot(range(len(d)), d,
                 marker='o', markersize=6,)

    # Create legend
    legend_labels = [weather_codes[i][1] for i in range(
   measure_counter)]
    plt.legend(legend_labels, title="Number of clouds",
               title_fontsize=14, fontsize=14
    )
    plt.title(f'{monitor} - GPU Load', fontsize=20)
    plt.xlabel('Runtime (seconds)', fontsize=15)
    plt.ylabel('FPS (Frames per Second)', fontsize=15)
    plt.show()


data = []
for i in range(len(file_path_comp)):
    data.append(prepare_data(file_path_comp[i]))

plot_cpu_data(data)
```

B.6 Installing applications with ADB

In order to install the application to the monitors a connection to the ADB service must be established by inserting the following command line on the CLI where the APK is located:

```
    adb connect <ip-address to the monitor>
```

Then to install the APK to the monitor the path needs to be specified or be in the same directory as the APK to transfer it to the monitor. This is done with the following command:

```
    adb install <apk>
```

If no other permissions are required the Android application should be installed to the monitor now and ready to be used.

B.7 Lighting

In order to replicate the effect of diffuse lighting on our 2D-cloud billboards, we get the light direction as a uniform value passed to the fragment shader. We then calculate the normal in the vertex shader as the initial normal (0,0,1) for all billboards but we then rotate it using the billboard's rotation matrix so that the normal rotates together with the billboard. This normal value is passed to the fragment shader. We then check if the dot product between this rotated normal and the opposite of the light direction vector is less than 0, in that case we flip the normal by multiplying it with -1. This ensures that if the light source is behind the viewer, the clouds will still be fully illuminated. The vector that goes from the current fragment's position to the billboard center is added to the previously calculated normal in order to further refine it. This effect ensures that when the light source is positioned to the side of the cloud, the side facing the light appears more illuminated than the side facing away from the camera, as the normals on the outermost part of the cloud are shifted the most towards the light source. The intensity of light is calculated by taking the dot product between the calculated normal and the opposite of the light direction to determine the angle of incidence. We take this dot product and add one to it and divide it by two so that the fragments with normals facing away from the light will get less illuminated instead of being totally dark. The code below shows the calculation of diffuse lighting:

```
    float ambient = 0.3;
    vec3 lightDirection = normalize(lightDir);
    vec3 norm = rotatedNormal;
    if (dot(normalize(cameraFront), -lightDirection) < 0.0) {
        norm = rotatedNormal * -1.0;
    }

    //Factor determining how much to change the normal vector
    float factor = 0.5;
    norm = normalize(norm + factor * (
        fragmentPos - billboardCenter
    ));
    float diffuse = (dot(normal, -lightDirection) + 1.0) / 2.0;
```

B.8 Gathering the data from the CPU and the GPU

The data will be gathered once the app has started running. We will collect data from the CPU using the built in functionality of the Linux command `top` using ADB. The data from the CPU will be the total load the application consumes on the CPU and is fetched through:

```
    adb shell top -b | grep cloud
```

where cloud is an alias for our application. This command tells the ADB shell to run `top` inside the monitors CLI, where this data will be collected in a text file later on.

The data of the GPU will be measuring the FPS (frames per second) which the GPU can handle. For the GPU measurement we have the following pseudo code in the application:

```
    static int frames = 0;
```

```
    static double lastPrint = now();

    frames++;

    if (lastPrint > 1.0) {
        auto cur = now()
        log("fps={}", (double)frames / (lastPrint - cur));
        frames = 0;
        lastPrint = cur;
    }
```

The code runs at every draw in the OpenGL code, in other words every time a frame is being rendered. A counter is then keeping track of how many frames have been rendered and checks if a second has passed and calculates the FPS. This FPS data is then written to a file inside the monitor's system. With ADB we fetch the FPS from the file after the test to measure how well the GPU handles the cloud arrangements and amount.

B.9 Optimized code for Experiments

```
    glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
    glBufferData(GL_ARRAY_BUFFER, cloud_cluster.numClouds *
        sizeof(glm::mat4), &modelMatrices[0], GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, indicesVBO);
    glBufferData(GL_ARRAY_BUFFER, cloud_cluster.numClouds *
        sizeof(float), &imageIndicesArray[0], GL_DYNAMIC_DRAW);
```

Listing B.4: Optimization done for Test case 3

EXAMENSARBETE Exploring Cloud Rendering Techniques for Aerospace Applications
STUDENTER Amjad Bakir, André Frisk
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Jacek Malec (LTH)

Utforskning av molnrenderingstekniker för flygapplikationer

POPULÄRVETENSKAPLIG SAMMANFATTNING **Amjad Bakir, André Frisk**

In-flight entertainment system i dagens flygindustri behöver uppgraderas för att ge en bättre användarupplevelse. Detta examensarbete undersöker prestanda med molnrendering på dagens flygskärmar för att undersöka om hårdvaran är tillräcklig och vad för optimeringar som kan användas.

På utvalda flygbolag kan flygplanet vara utrustat med in-flight entertaiment system i form av skärmar för att underhålla passagerarna under dem längre resorna. Mjukvaran för dessa system behöver ständigt uppgraderas för att ge passagerarna en god användarupplevelse som förstärker flygresan. Ett förslag på uppgradering är genom att framställa moln, även kallat för molnrendering, av verklighetstrogna moln med hjälp av väderdata där planet flyger. Väderdatan hjälper programmet att bestämma hur många moln som behöver renderas och hur de ska användas.

Vårt examensarbete undersöker hur dagens flygmonitorer (från Panasonic Avionics) kan hantera molnrendering med diverse optimeringar och hårdvarubegränsning. Genom att skapa en androidapplikation med olika effektiviseringar och antal moln kan man mäta prestandan från skärmarnas system för att undersöka hur mycket resurser molnen använder på systemet.

Vi använder oss av olika datorgrafiktekniker för att uppnå detta. Billboards, som är huvudtekniken som använts, är en teknik som ser till att objektet alltid är vinkelrätt mot kameran, som är ett virtuellt öga, utan hänsyn till kamerans rörelse. Detta simulerar ett 2D objekt som ett 3D objekt som ger verklighetstrogna simuleringar. Kombinerat med andra datorgrafiktekniker görs datamätning på skärmarna med hjälp av en egenbyggd androidapplikation.

Figur 1: Exempel från androidapplikationen med högt molnantal

Resultatet påvisar att dagens flygskärmar som används till passagerarna klarar av molnrendering till deras visualiseringsprogram vid lägre antal moln trots optimiseringarna som använts. Vid molnantal som överskrider 100 moln går prestandan ner avsevärt och kräver bättre optimeringar och lösningar för att visas. Resultatet påvisar även att det är möjligt att rita verklighetstrogna moln som speglar verkligheten med billboards som kan ge användaren en god verklighetsuppfattning om vad som sker utanför flygplanet.