Investigation of volumetric cloud rendering techniques

David Todd
b9052651
Video Demonstration: https://youtu.be/vhlTHSFk-Bc

Credit: Horia Varlan @ flickr

Abstract - This paper outlines how the rendering of volumetric clouds can be achieved software side, using ray marching techniques, and creates an implementation to test how this rendering can be optimized. The history of cloud rendering is investigated to show why a volumetric approach is necessary but computationally expensive, and a multitude of rendering variables are tested in the implementation to see what effects they have on performance, to make this technique as accessible as possible.

I. INTRODUCTION

The field of real-time computer graphics is a rapidly evolving area of interest when it comes to the creation of computer games, and other real-time computer applications. With an increase in hardware capabilities comes the ability to use more computationally expensive graphical techniques, and one particular technique that has gained significant popularity in recent years is the technique of ray tracing. The use of rays in computer graphics is a technique that has been discussed since 1968, though at the time it was noted that "Point by point shading techniques yield good graphic results but at large computational times."**[1]**. These large computational times persisted as the technique developed further, implementing global illumination methods for example. **Figure 1.**, published in 1980, took 74 minutes to render due to the hardware limitations at the time**[2]**.

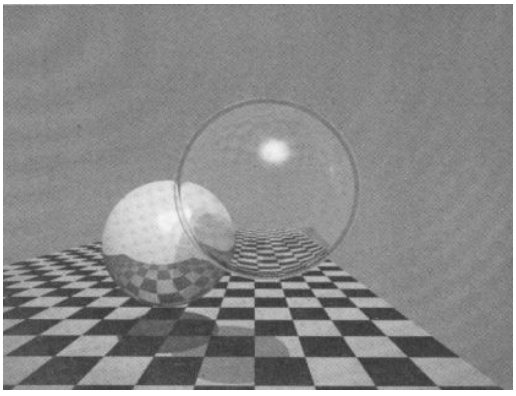

Figure 1. Image generated using ray tracing, displaying refraction.

Due to the extreme amount of time required to produce an image using ray based methods, ray tracing found a use in rendering premade cinematic scenes, but up until recently real-time graphics still relied mainly on rasterization, drawing objects in three dimensions before mapping to pixel space, to present a scene. Rasterization is typically a much faster process than ray tracing, however the method only focuses on one geometry primitive at a time, and this can contribute to a decrease in quality when compared to ray traced results. As mentioned by Dr. Philipp Slusallek, Director for Research at the Intel Visual Computing Institute, "Rendering with Rasterization is all but easy if you want to do anything more than drawing simple textured triangles. Too many PhD students have spent way too much of their time improving shadow tricks for rasterization -- and it still does not work correctly. The same is true for many other effects."**[3]**

With hardware capabilities increasing in recent years, ray tracing has seen a resurgence for the purpose of generating higher quality real-time images. In 2018 Nvidia announced RTX, a platform designed to accelerate ray tracing in applications, making use of dedicated ray tracing hardware**[4][5]**. While the cost of ray tracing on this hardware is greatly reduced, many devices still do not have this technology present, including consoles of the current 8th generation. For these devices, ray tracing must be done on the software side, and the performance cost is still greatly important.

There are many aspects of a graphical scene that can decrease the performance of ray tracing methods, such as a high number of reflections. This increases the number of rays generated, and therefore the number of intersection tests. However in the majority of scenarios, this is not a vital effect to have for a machine with lower hardware specifications. A more common issue for ray tracing in video games is the generation of clouds. Standard ray tracing cannot handle non-solid objects like clouds well, as there is no discrete intersection point. For this reason, an extension of the technique, known as ray marching, can be used. However, ray marching can have even greater impacts on performance than ray tracing, sometimes making the technique infeasible when combined with the resources needed by other aspects of the program.

This paper investigates the use of ray marching to create volumetric clouds, a graphical element that can be utilized by a wide variety of games. It will look at how clouds have been rendered previously with more limited hardware, and how volumetric clouds are now being implemented in industry projects. What have existing implementations had to do to maintain a high level of performance, and what are the positives and negatives of using this technique compared to others? Different approaches will be highlighted in the following pages, before a testing program is created to further investigate how different factors all contribute to the effect on performance that volumetric cloud rendering has, in particular when implemented on the software side.

II. RELATED WORK

Before investigating volumetric approaches to cloud rendering, it's important to understand what came before, and why a volumetric approach is necessary. 2D games will be left out from the analysis due to the fact that they do not try to realistically portray a world graphically, disregarding the 3rd dimension.

A. Super Mario 64 - Wing Mario over the Rainbow

This may appear to be an old case study, considering *Super Mario 64* was released in 1996, only recently after the introduction of 3D graphics to consoles, but the game actually implements the basics of three techniques still used to this day**[6]**.

The most obvious technique present in this level is to simply render a cloud as an opaque 3D object. This is an effective design choice to communicate where the platforms are, but it can be seen in **Figure 2.** that when Mario is standing on a cloud, he clips into the rendered object, in an attempt to show him "sinking" into it. This intersection between the Mario and cloud model is too

clean to appear realistic. Realism is the general problem with the opaque object approach, and while it can work for many titles, others require more realistic skies. It's also worth noting that if the camera were to travel inside the cloud, there would be nothing to display, save for the back of the cloud if backface culling was disabled.

Figure 2. Mario model intersecting with cloud model.

Also present in **Figure 2.** is the second technique, displaying clouds in the skybox, or drawing them before any other geometry and not altering the depth buffer. This approach certainly has a greater potential to look realistic, as gradients can be used to blend the clouds with the sky. In fact, research has managed to produce fantastic looking results that use this method**[7]**. However, the problem of intersection presents itself again, this time in a different form. Neither the camera nor any other object can ever intersect these clouds, making them completely intangible. There is also the issue of how cloud shadows are implemented when there is little sense of where the clouds are in relation to the world.

Finally, the third technique used by *Super Mario 64* to render clouds is present in Lakitu, the personification of the game camera. As seen in **Figure 3.**, Lakitu sits on a non-opaque, tangible cloud, that intersects with his model in a natural fashion. This is a good looking effect for the time, but it's deceiving. The "cloud" is actually made up of a collection of textured quads that implement billboarding so that they always face the player. Some of the quads are positioned behind the model while some are placed in front, producing the intersection effect.

Figure 3. Lakitu from *Super Mario 64.*

Once this is realized, the downsides of the approach become clearer. Notice how the colour of the texture never changes, only the transparency. With more detail applied to the texture, the collection of quads lose the 3D effect they currently portray. Additionally, as these quads are flat geometry, there is no simple way to calculate how their colour is affected by a light source.

Additionally, while it's not an issue in **Figure 3.**, intersections with geometry and the camera will still cause trouble. The billboarded geometry will intersect with an object or the camera in the same way the geometry would normally. This leads to unnatural looking intersection lines that don't match up with the texture on the geometry, and ruins the potentially more realistic look.

B. Mario Kart 8 - Cloudtop Cruise

Each of the three previously mentioned techniques has their advantages and disadvantages, but they can be and still are used effectively. *Cloudtop Cruise* is a modern example of using the three techniques together to create a scene full of clouds that doesn't impact performance**[8]**. Each of the aforementioned techniques can be improved by animating them, and *Mario Kart 8* has done that here with cloud quads that cover the track, as shown in **Figure 4.**. However, the issues regarding intersection still raise their head, and these quads only

appear when the camera is far away, fading as it gets close. The quads are also in fixed positions, leading to a very rigid cloud structure to avoid any graphical oddities.

Figure 4. Cloud quads cover the track, but fade out when the camera is near.

By still using rasterization, it's shown that good looking results can be produced, but there are plenty of restrictions on what can be done with them. If a cloud is to be affected by lighting, it must be a solid object, which does not look realistic. For a more realistic cloud, it cannot interact with other geometry without looking strange. No method for producing clouds by rasterization elegantly allows for the camera to intersect with the cloud. For all of these reasons, a volumetric approach must be considered.

C. Horizon: Zero Dawn

Due to the above reasons, some games in the 8th console generation have begun to implement volumetric approaches. Guerrilla Games attempted to use the traditional rasterization methods to create their cloudscapes, but found that it didn't meet their requirements as clouds were unable to evolve over time, and performance was impacted, both in terms of memory usage and overdraw**[9]**. As a high budget title, the goal was to produce the most realistic cloud visuals possible, and ray marching was employed to achieve this. The first notable improvement compared to rasterization is a memory saving technique, as all the cloud details in the entire sky are rendered using only three tiled noise textures. The textures are generated by combining two types of generation, perlin noise and worley noise.Perlin noise is a type of pseudo-random gradient noise that has been used extensively in computer graphics since it was introduced in 1985. This is due to the ability it has to mimic natural variation**[10]**. By scaling and adding layers of perlin noise together, more complex shapes can be produced. This is known as fractal noise, and is commonly used in volumetric cloud generation approaches**[11]**. However as noted by Schneider and Vos in their talk on *Horizon: Zero Dawn***[9]**, these clouds do not portray realism well enough. They state that the approach "has some nice wispy shapes, but it lacks those bulges and billows that give a sense of motion."

Adding bulges and billows to the cloud structure is the reason for the introduction of worley noise into the generation method. In contrast to perlin noise, worley noise creates textures better able to simulate the appearance of stone, water, and biological cells**[12]**. One generated worley noise texture is not very similar to a cloud structure, but there are a few modifications that can be made to improve the appearance. Inverting the texture creates the puffy, bubble like structure that is desired, and like perlin noise we can layer multiple generated worley textures to increase complexity. Combine this with layered perlin noise as shown in **Figure 5.**, and a satisfactory cloud generation texture can be produced**[13]**.

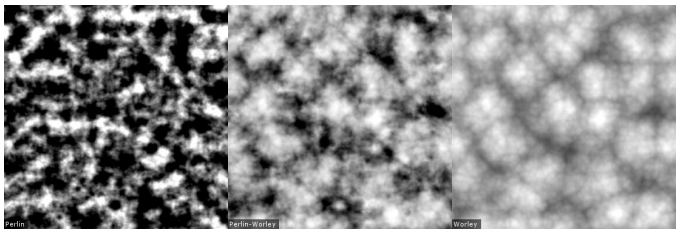

Figure 5. Perlin, Perlin-Worley, and Worley noise examples, all layered.

This perlin-worley noise is used as one channel of the first 3D texture used for cloud modeling, the other channels are all worley noise at different scales, and

together this texture is responsible for creating the base shape of the cloud. It may appear strange that perlin noise is only included in one of the channels, but as described before, too much perlin noise leads to clouds that do not have much structure to them. Other papers have also taken this approach**[14]**, so it's a proven technique for producing realistic looking formations.

An additional 3D texture is also used, containing three channels of worley noise at different frequencies. This texture is used to add detail to the cloud shape, and because it's only used for this purpose, this texture can afford to be lower resolution, and can be repeated more often. A repetitive look is avoided, as it's modifying an existing texture. Eventually however, the base texture will repeat and this will appear more obvious. To combat this, a 2D texture comprised of curl noise**[15]** is used to distort the shapes of the clouds, and helps to add realism when the clouds move, as with this texture present they will not all move in the same uniform fashion**[16]**.

In *Horizon: Zero Dawn*'s implementation of a cloud system, multiple other gradients were used to help calculate cloud structure. This was in an effort to make the clouds in the sky appear more true to life, separating out different types of clouds at the heights they would naturally occur**[17]**. This is not an approach that all games will want to take, particularly if only one variant of cloud is desired, so these gradients, and by extension sampling from them, are not essential to producing high quality cloudscapes. The cloud system also integrates with the weather system, darkening the clouds when rain is present. Again, this is a pleasant detail, but not essential for a cloud rendering integration. To discuss lighting the clouds, an academic source is better equipped to discuss general techniques.

D. Real-time rendering of volumetric clouds - Fredrik Haggström

The approach presented here is similar to that of *Horizon: Zero Dawn*, in terms of both modelling and lighting methods**[14]**. However, what sets this approach to lighting apart is the attention paid to optimization throughout. Lighting the clouds involves ray marching, and will be the most performance intensive element of the visual effect, so any performance gains that can be used are essential to ensure that the effect can feasibly be rendered on lower power hardware.

In the real world, light energy comes from a light source, is reflected off objects, and a portion of that reflected light reaches a human eye. When using ray tracing, this process is essentially reversed, with rays being cast towards objects from the camera, and then once the object is reached having new rays cast to simulate colour information, shadows, reflections, and plenty more visual effects, as seen in **Figure 6.**. With solid objects this is able to produce a very realistic look, as reversing the direction of the light rays when compared to how it really operates doesn't affect much. However with non-solid objects like clouds, this is not the case. This is due to the refraction of light, and while for solid objects that refract light like glass there are distinct ways to model how the light rays will travel, for a cloud there is no easy way to model the extreme refraction of light that takes place.

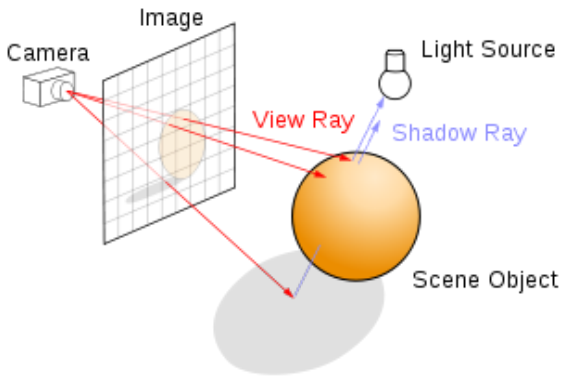

Figure 6. Diagram showing standard ray tracing on a solid object.

Clouds are made out of millions of water droplets, so when light reaches a cloud, it bounces around an innumerable amount of times before potentially reaching an eye. Obviously casting rays to represent all that refraction is incredibly wasteful, so instead ray marching is implemented. The idea behind ray marching is to

move through the cloud in steps, sampling its density (which in this case can be represented by the aforementioned textures) to give an idea of how much light will be able to reach an eye, and how much will be refracted away. This is modelled by a function applied to the density with Beer's Law, as seen in **Equation 1.**, being a popular choice due to how it models the attenuation of light as it travels through a material**[18]**.

$$Intensity = e^{-density \times constant}$$

Equation 1. Beer's law when applied to the cloud problem.The constant is for artistic effect.

Of course, to actually calculate how much light reaches each of the sampled points, we must also cast a ray from each sampled point. And as we are traveling through a non-solid object, again ray marching must be used, applying beer's law again while sampling densities, as shown in **Figure 7.**. This is where the true performance cost of cloud rendering lies. While not entirely accurate, ray tracing could be compared to a $O(1)$ problem, whereas ray marching through a cloud would be $O(n^2)$.

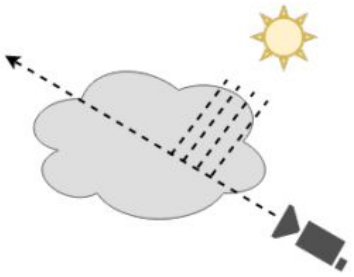

Figure 7. Raymarching through a cloud. Samples are taken all along the rays generated.

Thankfully, there are a host of optimizations that can be implemented to improve the performance of this ray marching method. First is to stop marching once the cloud density is high enough that further away light won't be transmitted. This saves plenty of texture samples that wouldn't have any effect on the final result from the ray. It's also safe to check if there's any opaque geometry in the ray's path, and if so stop marching once that point is reached.

Haggström's implementation also details a technique that allows the step size to be increased. Increasing the step size means less sampling, but offers less realism and can result in a banding effect when the step size is too high. However, by varying the start position of the ray march using blue noise, this banding is eliminated due to sampling no longer occurring on planes. Another technique to increase the step size is to multiply it by a factor when the density sampled does not meet the threshold to qualify as a cloud. Due to the smooth nature of worley and perlin noise, the difference between the sampled density and the threshold can be used to calculate how empty the sky is in the current area. If a cloud isn't expected for a while, the step size can be greatly increased until the threshold is met, and the step size returns to normal.

III. DESIGN AND IMPLEMENTATION

A. Tools Used

To render clouds and accurately measure performance and quality, a program was created using C++ and OpenGL, dedicated to the rendering of clouds and factors that may affect performance. The use of C++ and OpenGL was to start from as close to scratch as possible within the scope of the project. This avoided the use of a game engine, or any other inclusion that could result in performance overheads, that could skew the results or introduce other unintended consequences. This is also why only one extra piece of middleware was used, ImGui**[19]**. ImGui allows for the easy introduction of UI elements that were essential when testing various settings of the rendering process. ImGui is also not very performance intensive, so the additional resources required to display the user interface would not dramatically affect any frames per second readings taken.

B. Measuring Quality

There are plenty of factors that contribute to the performance of ray marching cloud rendering techniques, but many of these factors can also affect the quality of the render produced. The trouble with analysing the quality of cloud images produced is that quality is subjective and there is no objective way to measure how much a rendered image looks like an image of a different cloud. For that reason, a baseline was established for what could be considered a desirable result, and every other result was compared to that.

To create the baseline result, a cloud was rendered with all the settings set as quality-oriented as possible. Captures were then taken at various positions, and then when other settings were applied the same captures were taken again. These captures were then compared to the baseline captures, and given accuracy scores based on the structural similarity index**[20]**, and the mean squared error between the two images. Performance was measured by the average frame rate over 5 seconds in the position where each capture was taken. Averaging over this timeframe was chosen for a few reasons:

- Anomalous frames had a reduced impact
- It made it easy to tell immediately whether a variable change was increasing or decreasing performance
- ImGui already has an implementation for this method**[19]**, avoiding the situation where readings are incorrect due to programming errors

Results were then analysed to establish the effect on both render quality and performance, to help draw conclusions on where the most beneficial tradeoff was.

C. Generating 3D noise

To allow the clouds to move over time, there would need to be changes to the texture that cloud density was sampled from. However, creating a 3D texture every frame would be far too expensive computationally, so the method of using multiple textures that are offset as time goes on was used, similar to existing implementations**[14]**. For this to work successfully, the texture needed to be seamless so that it could be repeated endlessly. Additionally, the noise generated should be reproducible for testing purposes, and therefore not totally random. It should also be reasonably fast to create the textures, as extending load times is not desirable.

There were multiple techniques used to achieve all of the above. First, to improve creation times, the texture was split into an $n \times n \times n$ grid of cells, and a worley point was created pseudo-randomly within each cell. This meant that when it came to calculating the fragment values for the texture, rather than comparing the location to each worley point to find the minimum distance, it was only necessary to check the points in the current and adjacent cells. In this case, the 3x3x3 grid around each fragment. This method of optimization was derived from an existing unity implementation**[21]**, which saw significant speed increase for texture generation with this method.

Handling generation this way also made creating a seamless texture easier, as fragments in cells at the edge of the texture could simply compare with the cells on the opposite side. Without this optimization, to achieve a seamless texture the entire set of points would need to be duplicated 26 times, to surround the original 3D texture. This would drastically increase the number of comparisons each fragment would have to make, but with the optimization this was no longer the case.

It was also important to ensure that the random values were not actually random. This was done with the use of a hash function, described in **Equation 2.**. The matrix M could be swapped out to change the random values, and this was done for each channel of the image.

$$frac(28.9 \times cos(v \times M))$$

Equation 2. Pseudo-random hash function, with v as the input vector

Perlin noise was implemented in a similar fashion, but was only included in one channel to avoid overly wispy

clouds. Not included in the final version was Curl noise**[15]**, as it was felt it too drastically changed the output, and made analysis of results more difficult while not affecting performance in any meaningful fashion. While it would be useful for larger cloudscapes, this was not the situation for it.

D. Tested Variables

Clouds were rendered using both fragment and compute shader methods. Implementing this was relatively simple, as the majority of the shaders were the same. At the end of the compute shader, the result was saved to an image that was then displayed using a textured screen wide quad, while the fragment shader drew to the screen quad directly.

Figure 8. Typical result from the project program.

The fundamentals of marching through the cloud work similar to existing solutions, and the results look rather impressive, as shown in **Figure 8.**. The clouds in this figure have a similar appearance to dense cumulus clouds**[22]**, though using the multiple structure settings available plenty of other cloud type visuals are achievable. The modifiable structural settings are as follows:

- Cloud Scale - How the density noise texture is scaled before being sampled from
- Detail Scale - Identical as Cloud Scale, but relating to the detail texture
- Cloud Speed - A multiplier applied to the sampling offset, that increases as time goes on
- Detail Speed - Identical as Cloud Speed, but relating to the detail texture
- Density Multiplier - Larger values result in thicker clouds, with a negative value inverting the texture
- Density Offset - What density value constitutes a cloud, with higher values meaning less clouds
- Cloud Boundaries - The endpoints of the AABB that the clouds are kept within

Clouds are rendered in an axis aligned bounding box (AABB) structure which, due to the simple nature of a ray-AABB intersection test, avoids time spent doing ray calculations outside of the potential cloud space. This is in contrast to some of the approaches looked at earlier, which use a spherical atmosphere, to simulate the curvature of the earth**[9][14]**. The decision was made to deviate from this approach, partly due to the fact that analysis would be easier with an AABB, but also due to the fact that not every game developer would want to render their clouds in a spherical atmosphere, in particular if the game involved the player interacting with, or being in the same vicinity as the clouds.

In addition to the ray marching step sizes, there are some additional lighting variables. The most notable of these are the variables related to light scattering, which involved the inclusion of the Henyey-Greenstein phase function**[23]**, which was part of the pre-calculated phase value added to the lighting calculation. This function helps model the silver lining effect in clouds facing the direction of the sun. Speaking of the sun, the lighting direction can be changed, alongside the sun, sky, and cloud colour.

In a further attempt to optimize the program, an option called “Step Optimization” was developed. When an area of the texture was sampled far below the density offset, the step size was increased by a user defined factor, as the low sampled value, and the gradual nature of the noise texture, meant there were definitely no clouds nearby.

IV. RESULTS AND EVALUATION

Testing was carried out across multiple resolutions, on multiple different devices. As the program generated the same images on each device, measures of image quality remained the same across all configurations. However, due to the different hardware and resolutions being used, FPS results varied from one configuration to the next. This wide approach to testing was done to ensure that the results were generalized, and would likely be the same on any other piece of hardware. Because of the disparity in FPS, little attention was paid to the actual values, instead focusing on how the data changed as the variables were adjusted. For the majority of tests, hardware made little difference to the final conclusion, but more detail is provided for the tests where there were irregularities between hardware results.

The GPUs tested on, and maximum resolutions, were as follows:

- NVIDIA GeForce RTX 2080 Ti OC with 11GB GDDR6 (1920 x 1080)
- NVIDIA GeForce RTX 2080 Super with 8GB GDDR6 (2560 x 1440)
- NVIDIA GeForce GTX 1070 with 8GB GDDR5 (2560 x 1440)
- NVIDIA GeForce GTX 960 with 4GB GDDR5 (1920 x 1080)
- NVIDIA GeForce 720M with 2GB GDDR3 (1280 x 720)

Unless otherwise mentioned, the figures in this section relate to readings taken using the "View 2, Clouds 1" configuration. Conclusions drawn from these results were similar to the majority of other configurations tested. Where this is not the case, more detail will be given on the outlying configurations.

A. Raymarch Step Size

The first variable investigated was the variable hypothesized to have the largest impact on performance, the step size of the ray march from the camera to the cloud. **Figures 9. and 10.** shows the plots for average frames per second (FPS), and mean squared error (MSE) of the images produced, as the step size variable was altered.

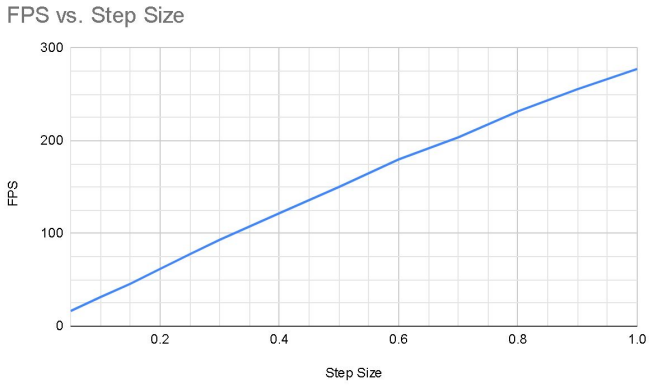

Figure 9. Average frames per second plot

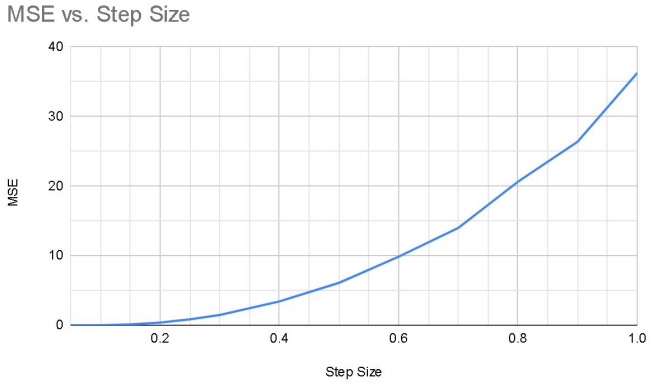

Figure 10. Mean squared error plot

For the MSE readings, the image produced with a step size of 0.05 was used as a comparison, as it was the image that took the most samples and produced the most accurate result of what the image would look like with an infinite number of samples. With this in mind, results pointed to the conclusion that as step size increased, MSE rose in an exponential fashion, and therefore image quality decreased in an exponential fashion. At the same time, FPS increased in more of a linear fashion. These two facts combined meant that the ratio between performance increase and quality decrease got worse and worse as step size was increased.

Within the context of game development, to strike a happy medium between performance and quality, the solution in this instance appears to be targeting a specific frame rate, and decreasing the step size to the point where that frame rate is met. With this data, to target 60 FPS, an industry standard, a step size of around 0.2 would be appropriate.

MSE was not the only measure of image quality taken. As MSE does not take into account the structural properties of an image, a structural similarity index measurement (SSIM) was also taken**[20]**, alongside a multiscale variant**[24]**. This multiscale variant value (MS-SSIM) was chosen due to it's higher accuracy compared to SSIM, allowing it to be a measure for more subtle changes within the clouds. **Figure 11.** shows the results.

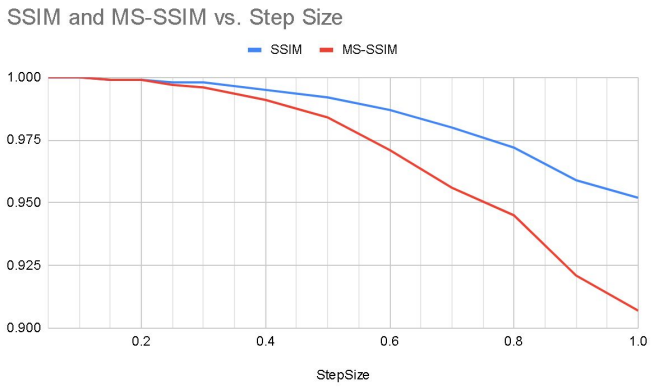

Figure 11. SSIM and MS-SSIM plot

Worth noting is the fact that the lowest value in these figures is still above 0.9. However, that isn't too surprising when considering the fact that the clouds only consist of part of the image, while the rest remains unchanged. The important takeaway is not the values themselves, but the fact that like the MSE plot, the data points to image quality decreasing at an exponential rate. Unlike MSE however, the SSIM and MS-SSIM values are bounded between 1 and 0, and there's only so much degradation in structure that can take place. For that reason these values will not continue the exponential decrease with larger step sizes, but instead level out. The beginnings of this can be seen in the readings for a step size of 1.0. It's worth mentioning that once the step size reaches a certain length, the samples occur at very sparse intervals, and an SSIM value isn't really representative of how clearly incorrect the result looks, as shown in **Figure 12.**

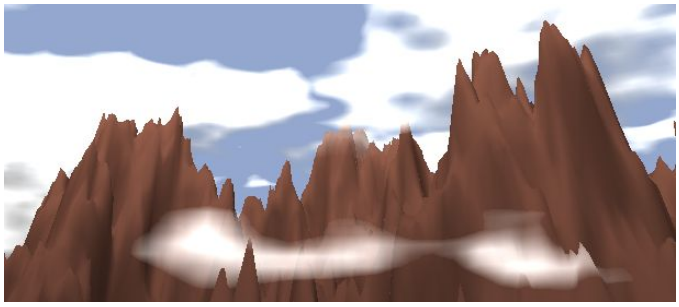

Figure 12. Step Size of 5.0, notice the total lack of cloud lighting.

B. Lighting March Step Number

Unlike the previous variable from camera to cloud, the variable to control step size from cloud to light source defines the total number of steps, rather than the step distance. This is to prevent clouds near the top of the cloud AABB taking less samples and getting less accurate results. The downside of this method is that the image quality is affected by the direction of the light source, as if there is a far distance to travel through the cloud box to get to the light, samples are spread very thin and results are not accurate at low step numbers.

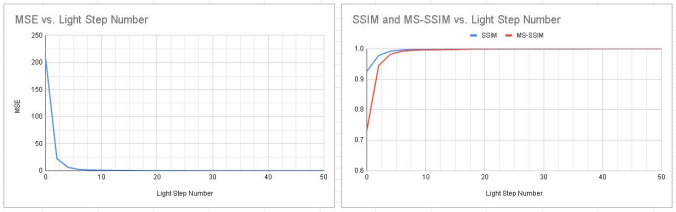

Figure 13. At a certain point, additional light steps are unnecessary

This makes any general conclusion rather difficult. As seen in **Figure 13.** there is a point where after the quality improves rapidly, it reaches very close to perfect levels. This is an obvious point to set the light step number to,

but this value is different for each combination of light direction, and cloud AABB size. This means that deciding on a value for this variable requires testing by the developer for their specific situation, to achieve the best optimized value that keeps the majority of visual information.

FPS is affected in a similar fashion to the camera to cloud step size, though as the step number does not change the step length in a linear fashion, the FPS value does not change in a linear fashion either.

C. Step Optimization

The addition of step optimization was designed to reduce the number of samples taken in areas of the cloud box where there were no clouds nearby. For that reason, it was surprising when results were consistent among varying cloud types. Even configurations involving "Clouds 3" which covered the cloud AABB with dense clouds, benefitted from Step Optimization. Increase rates of FPS were consistent across all tested devices, though they were not quite as drastic as had been hypothesized. **Figure 14.** shows the results for the "View 2, Clouds 1" configuration.

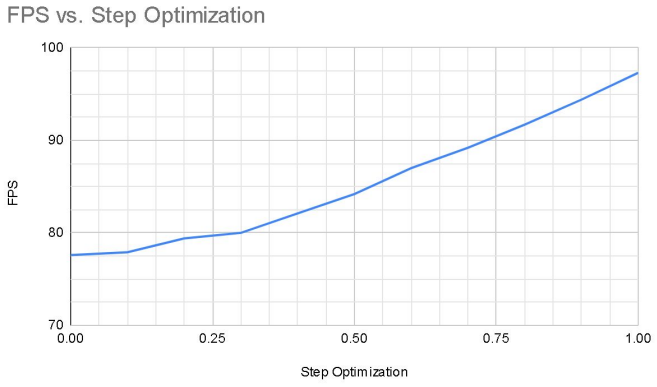

Figure 14. Average frames per second plot

There's a slight curve to the graph as Step Optimization increases, but after 0.5 the increase becomes rather linear. Meanwhile, the MSE, SSIM, and MS-SSIM values appear to again be increasing/decreasing at an exponential rate. However, unlike changes to the step length, this is not as big a problem, as the values themselves are far closer to the original images, as can be seen in **Figures 15. and 16.**

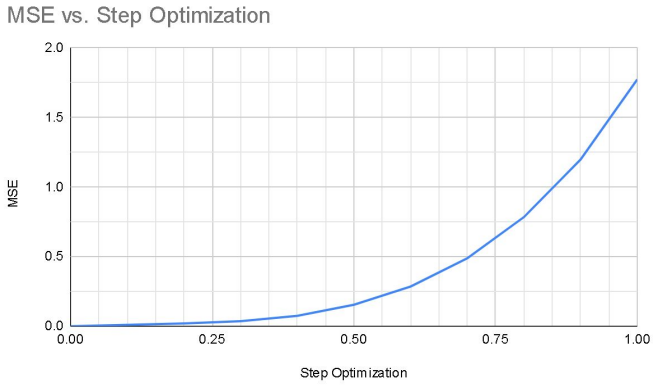

Figure 15. Mean squared error plot

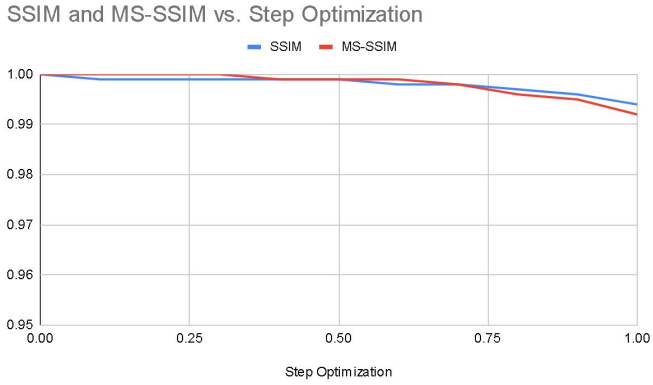

Figure 16. SSIM and MS-SSIM plot

This suggests positive results, as it seems a Step Optimization value of 1.0 does not impact the cloud visuals too heavily. However, tests were only carried out up to a value of 1.0 for a reason that can't be captured by analysing the image produced.

The deviations from the best quality versions of the image vary depending on the location of the camera.From outside the cloud AABB this isn't an issue, but moving into or close to the cloud box reveals strange visual artifacts at the box edges, and a wobbling effect as the camera moves through and samples different areas. This is a very distracting, unnatural looking effect, and

means that the majority of the time while inside the clouds, a Step Optimization value of around 0.4 is the highest possible value that doesn't introduce wobble. This "safe" value also changes depending on cloud structure. Unfortunately as shown by the FPS graph, these lower values aren't getting the most out of the FPS increases, so while Step Optimization is moderately useful in all cases, it only really excels when the camera is far away from the cloud box.

D. Fragment vs Compute Shader

Where the majority of tests produced similar results independent of hardware, testing performance based on the shader used produced a notable anomaly. On all hardware tested but one, using a compute shader to render clouds was significantly faster than using a fragment shader. Average FPS results for the highest performing hardware can be seen in **Figure 17.**

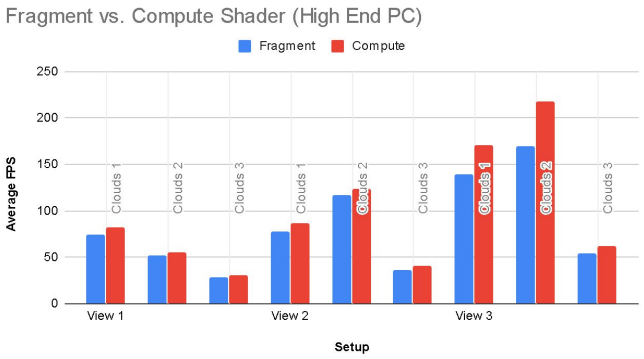

Figure 17. Fragment vs. compute shader results on an NVIDIA GeForce RTX 2080 Ti GPU

Frame rates are on average, around 1.2x higher when using a compute shader, likely because the compute shader does not need to go through the rasterization pipeline. These results are impressive, especially considering that the exact same visual output is produced. However, there was an outlier in these results in the form of the FPS readings from the NVIDIA GeForce 720M, the lowest power GPU tested on. Those results are shown in **Figure 18.**

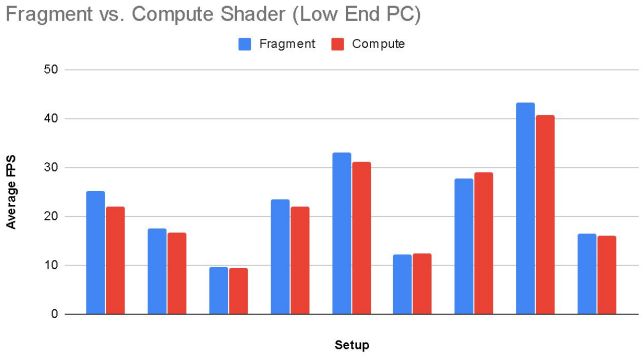

Figure 18. Fragment vs. compute shader results on an NVIDIA GeForce 720M GPU

Results on this hardware are far more mixed when making comparisons between the types of shader. Generally the fragment shader appears to be faster, but that's not consistent across all locations the clouds are viewed from, as seen in the "View 3 Clouds 1" configuration. Other locations where the compute shader outperformed the fragment shader were found in more informal testing. The reasons for these fluctuations were unclear, as were the reasons that the compute shader was generally slower on this hardware. However, there was a theory on why this was the case.

It's likely that due to the dated nature of the GPU, it's not fully optimized to make use of compute shaders, a comparatively new introduction to the field of computer graphics. On the other hand, fragment shaders have existed for much longer, so on older hardware more optimizations will be present for fragment shaders.

Additionally, on this lower end PC, other factors are also heavily contributing to the frame rate. As an example, removing the mountains in the "View 1 Clouds 1" configuration increases the FPS from 25.1 to 27.2, an increase of 8%. While tessellation is present here, the mountains are not an intensive aspect of the program. The fact that removing them causes a larger proportional change in FPS than changing the type of shader points to the fact that for this hardware, changing the shader has an almost negligible effect on performance. This leads us to the conclusion that in general, a compute shader is

preferable for volumetric cloud generation, although for lower powered hardware a compute shader may perform worse than a fragment shader in some cases. However, at the level this is occurring, the difference between compute and fragment shader performance is almost negligible.

E. Cloud Structural Changes

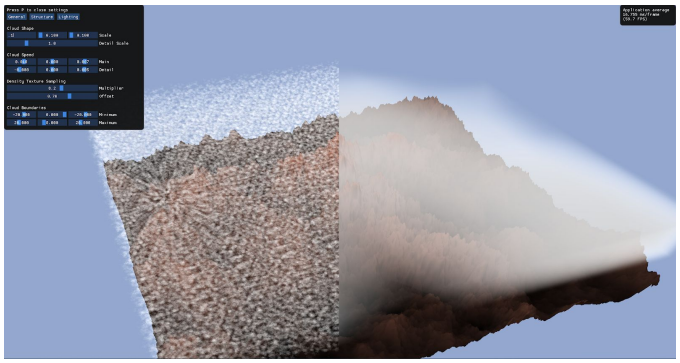

Figure 19. Clouds 1 configuration at scale 0.1 and 50 (left and right respectively)

As shown in **Figure 19.**, changing the scale of the clouds produces dramatically different results, but the total overall coverage is approximately the same in this case. This is interesting because testing reveals that when clouds are rendered at the smaller scale, the FPS drops sharply, as seen in **Table 1.**. The result for later scales of Clouds 2 is a result of very few, or no clouds at all being present in the scene past a scale of 10.

Cloud Scale	0.1	0.5	1	5	10	50
FPS for Clouds 1	14	100.6	156.3	154.2	84.5	60.1
FPS for Clouds 2	17.2	143.4	159.3	226.9	531	604.2
FPS for Clouds 3	4.7	42.6	55.1	57.7	58.5	62.1

Table 1. Average FPS vs. Cloud Scale for the three cloud testing configurations

While covering the same amount of area with cloud, the way each scale applies this coverage is what causes the large disparity between frame rates. In the scenario with the large singular cloud covering the mountains, rays march through the cloud, but because it's so large and dense, the transmittance level quickly drops below the threshold for making a visible difference, and the code exits early, as progressing the ray through the box further will change nothing.

This is not the case with the smaller scale scenario. Here, because of the pockets of air between the smaller clouds, the transmittance does not drop as quickly and rays must travel further through the AABB, resulting in more samples being taken. Thanks to how cloud lighting works, by sending out even more rays, the number of samples expands drastically. Sampling is by far the most computationally expensive operation in the shader, and the additional amount of samples needed to generate the extra cloud lighting cripples the frame rate. It can be proven that this is the cause by increasing the size of the AABB, where frame rates reach similar levels at higher cloud scales, because again for rays to get through the box they need to pass through multiple clouds with gaps in between.

F. Light Scattering

Due to the way light scattering was implemented, it produced no significant additional performance cost. This was because rather than calculating the phase value at the density sampling step, the phase value was pre-calculated upon the creation of each ray from the camera. This led to the function being called far less than it would do if it was part of the sampling process, no doubt saving a large amount of FPS as a result.

Forward or back scattering is an artistic choice, so measures of image difference don't provide information that any conclusions can be drawn from. There's also no way to decide upon what setting should be used as a point of comparison, as no particular setting looks objectively better/more detailed than another. **Figure 20.** shows the difference between having scattering on or off by changing the "Phase Factor" variable. Whether scattering is useful or not will vary from game to game.

Figure 20. Phase Factor at 0.95 and 0.0 (left and right respectively)

V. CONCLUSIONS

There are certainly situations that have not been analysed by this paper. Cloud performance is based heavily on the location that it's viewed from, and it's impossible to cover and analyse every context that clouds could be viewed in. That said, many common situations have been analysed, and some general guidelines can be drawn from the results produced.

Firstly, it's clear that so long as you're not on a low spec GPU, compute shaders will be the preferable way to generate volumetric clouds. If you are on a low spec GPU, then volumetric clouds will be just one factor amongst many that will decrease the performance of a game, so rather than simply attempting to use a fragment shader instead, it may be more appropriate to search for an alternative solution if the clouds are not an integral part of the experience. As discovered in the research phase, there are many methods for generating cloud visuals, so it's likely that for any game there is another cloud generation method that, while it may not be as flexible, will help the performance of the program far more than volumetric clouds will.

That said, there are still plenty of ways to optimize volumetric clouds without sacrificing much in the way of visual quality. The most significant of these appears to be changing the step size in the camera to cloud ray march. With careful consideration and testing from developers, the step size can contribute to large performance savings without affecting image quality to a noticeable degree. This extends to the cloud to light ray march step size, although in this case even more attention needs to be paid to the context in which the clouds will be viewed. In a game with a time of day mechanic, the sun will likely appear in different positions, which will affect how many lighting steps are required.

Step Optimization is also a very useful technique, that in all tested scenarios can improve performance without sacrificing image quality. The only catch is that the gain in performance is marginal compared to other factors, though the fact that it does improve how quickly a program runs makes it remain a positive addition to any volumetric cloud rendering program.

In contrast to techniques that don't change much visually, changing the shape and structure of clouds is, perhaps unsurprisingly, the biggest factor overall affecting how resource intensive the cloud system is. As finding a cloud when sampling the noise texture creates more work for the ray to do, it follows that the more clouds you have on screen, the slower the system will be. This brings us back to the developer using volumetric clouds in their game, who should factor in the cloud coverage/performance trade off when implementing a similar system.

As with many areas of computer graphics, the final result could be improved in a multitude of ways. There are plenty of extensions that could be made to this project, but the most interesting from a performance perspective would be the use of downsampling. Rendering the clouds at a lower resolution before upscaling the result would definitely save performance. This does lower the image quality, but not to a noticeable degree, especially on higher resolution monitors. Part of **Figure 21.** has been edited to show what this optimization would look like. Or rather, what it wouldn't look like. Especially with the use of forward scattering, which removes detail anyway, very little is

different to the standard method which in this case, would use approximately 4 times as many rays.

Figure 21. The lower half of the image is downsampled, making every 2x2 group of pixels identical. Post processing would improve the visual here even further.

It's also worth bearing in mind that this method only needs to affect the clouds, and buffers can be set up so that all other visual elements are unaffected. The method also doesn't need to go so far as to cut out the concept of rays relating to a single pixel. While not as kind to performance, one could engineer a system where a camera to cloud ray is sent out for every 3x3 area of pixels, with offsets to calculate where the cloud lighting rays are sent from for each pixel. This starts getting into complex optimizations regarding compute shaders and memory sharing outside the scope of this report, but it's a definite optimization avenue that could be explored.

The system also lends itself well to more functional extensions. Most obvious is the shadows from clouds affecting other objects, in the case of this program the mountains. This would require extending the project to handle geometry to light source rays, that pass through the cloud AABB and take samples much in the same way as the already existing cloud to light source rays.

Another useful addition would be the changing of the cloud area shape. As mentioned in the *Horizon: Zero Dawn* cloud presentation**[9]**, a cloud area made up of the difference of two large spheres better represented the spherical nature of earth, and the lowering of clouds towards the horizon. This would be relatively simple to implement, as ray/sphere tests are simple and cheap, and the spheres would presumably move along with the player, at least on the X and Z axis. The only challenge would be modifying how the 3D noise texture is sampled, to simulate the texture curving round the sphere.

VI. FINAL THOUGHTS

With the introduction of hardware specifically designed to handle ray computations, volumetric rendering is set to become even more prevalent in computer graphics. This makes finding techniques that can optimize the process, whilst sacrificing the minimal amount of detail, more important than ever, especially considering that many devices will still be performing ray casting logic software side. Rendering volumetric clouds in real-time is a computationally expensive task, but the results are more impressive than anything that has come before, and with the right optimizations it's possible on the majority of modern GPUs. With that said, volumetric clouds are not the only cloud generation technique worthy of consideration. Looking back at the implementation of clouds in *Mario Kart 8***[8]**, it's effective due to using multiple different techniques at once, so the flaws of each method are covered up by the others. Likewise, during testing and showcasing the program to others, the majority of people agreed that adding a cloud filled environment map would improve the visuals even further.

That's the key thing to remember. Volumetric clouds are very impressive, and are able to be rendered cheaply enough on the majority of modern hardware, but they're only one tool in the cloud maker's toolkit. With good art direction, multiple cloud techniques in play, and other aspects of presentation polish, volumetric clouds can be more than just a realistic simulation, and reach truly impressive heights.

VII. REFERENCES

[1] Appel, A. 1968. Some techniques for shading machine renderings of solids. Proceedings of the April 30--May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring).

[2] Whitted, T. 1980. An improved illumination model for shaded display. Communications of the ACM 23, 6, 343-349.

[3] Slusallek, P. and Kirk, D. 2020. Raytracing vs Rasterization. Scarydevil.com. http://www.scarydevil.com/~peter/io/raytracing-vs-rasterization.html.

[4] NVIDIA RTX™ platform. 2020. *NVIDIA Developer*. https://developer.nvidia.com/rtx.

[5] James, P. 2020. GDC 2018: NVIDIA Announces 'RTX', Real-Time Ray Tracing Engine for Volta GPUs. *Road to VR*. https://www.roadtovr.com/gdc-2018-nvidia-announces-rtx-real-time-ray-tracing-engine-volta-gpus/.

[6] Nintendo64Movies. 2013. Super Mario 64 Walkthrough - Wing Mario over the Rainbow. https://www.youtube.com/watch?v=zfXO8JygQjI.

[7] Mukhina, K. and Bezgodov, A. 2015. The Method for Real-time Cloud Rendering. Procedia Computer Science 66, 697-704.

[8] GamerGJB. 2014. Mario Kart 8: Cloudtop Cruise [1080 HD]. https://www.youtube.com/watch?v=qyAI-i3iOBk.

[9] Schneider, A. and Vos, N. 2015. The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn. .

[10] Perlin, K. 1985. An image synthesizer. Proceedings of the 12th annual conference on Computer graphics and interactive techniques - SIGGRAPH '85.

[11] Clouds using 3D Perlin noise. 2020. Shadertoy.com. https://www.shadertoy.com/view/XlKyRw.

[12] Worley, S. 2020. A Cellular Texture Basis Function.

[13] Fewes/CloudNoiseGen. 2020. GitHub. https://github.com/Fewes/CloudNoiseGen.

[14] Fredrik Haggström, F. 2020. Real-time rendering of volumetric clouds.

[15] Bridson, R., Houriham, J. and Nordenstam, M. 2007. Curl-noise for procedural fluid flow. ACM Transactions on Graphics 26, 3, 46.

[16] randomTube80. 2017. From Guerrilla-Games "Volumetric Cloudscapes of Horizon Zero Dawn" presentation.

[17] Cloud Types | UCAR Center for Science Education. 2020. Scied.ucar.edu. https://scied.ucar.edu/learning-zone/clouds/cloud-types. https://www.youtube.com/watch?v=FhMni-atg6M.

[18] Bouguer. 1729. Essai d'optique sur la gradation de la lumière. Pl. 3. Paris.

[19] ocornut/imgui. 2020. GitHub. https://github.com/ocornut/imgui.

[20] Wang, Z., Bovik, A., Sheikh, H. and Simoncelli, E. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. IEEE Transactions on Image Processing 13, 4, 600-612.

[21] SebLague/Clouds. 2020. GitHub. https://github.com/SebLague/Clouds.

[22] Met Office. 2020. Cumulus clouds. https://www.metoffice.gov.uk/weather/learn-about/weather/types-of-weather/clouds/low-level-clouds/cumulus

[23] Henyey, L. and Greenstein, J. 1941. Diffuse radiation in the Galaxy. The Astrophysical Journal 93, 70.

[24] Abdel-Salam Nasr, M., AlRahmawyb, M.F., Tolbab, A.S. 2016. Multi-scale structural similarity index for motion detection. Journal of King Saud University 29, 399-409.