Aetherius: Real-Time Volumetric Cloud Generation Tool for Unity

Oscar Pérez Martín

Centre de la Imatge i la Tecnologia Multimèdia, Universitat Politècnica de Catalunya

Bachelor's degree in Video Game Design and Development (2021-22)

Mr. Marc Garrigó Invers

June 30, 2022

Aetherius © 2022 by Oscar Pérez Martín is licensed under CC BY-NC-SA 4.0

Abstract

This thesis describes the development of Aetherius, a Unity tool which can generate and visualize virtually endless and unique cloudscapes in real-time dynamically; The resulting tool can be used in videogames to easily and quickly create immersive and dynamic skies without wasting resources in the development of a dedicated system.

Developing a volumetric cloud system is complicated and especially small studios do not have the resources to create such systems for their skies. The objective of this project is to provide an accessible and easy to use alternative for small studios and indie developers to turn static, boring and featureless skies into high quality ones.

In this document the problems encountered during the development of the tool and the techniques used to generate, render and optimize cloudscapes are described; to test the tool's usefulness this project includes the creation of a small demo application.

Acknowledgements

I would like to acknowledge everyone that helped me get to where I am now, for their support and inspiration during the development of this thesis and project.

I want to thank my friends and my family for being there, aiding me when I needed it and helping me test the project. I would also especially like to thank my mom Silvia and my friend Carla for being as enthusiastic as I am about clouds and sending me photos of them every time they looked at the sky and saw cool cloudscapes. Finally, I want to thank Adrià Serrano for helping me with mathematical questions when I needed it.

Keywords

Ray Marching, Volumes, Clouds, 3D, Procedural Generation, Real-Time, Unity, Tool, HLSL, Videogames

Links

The Unity project containing the source code and files for the tool in this thesis can be found on GitHub through the following link:

https://github.com/oscarpm5/Aetherius

The different demonstration applications that have been compiled for the project can be found in the same repository under the Releases section or through the following link:

https://github.com/oscarpm5/Aetherius/releases

The trailer showcasing the tool can be found at:

https://youtu.be/OHnivbkmO6s

Table of Contents

I. STATE OF THE ART ... 16
Cloud Representations ... 16
Skybox ... 16
Billboard ... 17
Polygon ... 18
Voxel ... 19
Procedurally Generated Clouds ... 19
Volumetric Rendering ... 20
Volumetric Cloud Tools Available ... 23
Unity Engine ... 25
Unity Packages ... 26
Unreal Engine ... 28
II. METHODOLOGY ... 30
Documentation Structure ... 30
Procedure and Tools for Project Monitoring ... 30
Gantt with Agantty ... 30
Kanban with Trello ... 30
Version Control Tools with GitHub and GitHub Desktop ... 32
Evaluation Methods ... 33
Objectives Validation ... 33
Task Validation ... 33
Risks and Contingency Plans ... 34
General Risks ... 34
Concrete Risks ... 34
III. PLANNING ... 36
Phases of Development ... 36
Pre-Production ... 36
Production ... 36
Post-Production ... 38
Initial Cost Analysis ... 38
SWOT Analysis ... 40
Planning Changes and Deviation (15/03/22) ... 41
IV. DEVELOPMENT ... 42
Procedural Noise Generation ... 42
Worley Noise ... 43
Improved Perlin Noise ... 45
Fractal Brownian Motion ... 47
Cloud textures ... 48
Weather Map Textures ... 51
Cloud Modelling ... 51

Ray March 51
Density Model 53
Weather System 57
Wind and Skew 57
Cloud Layers 57
Presets and Transitions 58
Cloud Lighting 59
Multiple Scattering Approximation 64
Day / Night Cycle 64
Scene Integration 65
Object Occlusion 65
Banding Reduction 65
Atmosphere 66
Custom Editor 68
Optimization 68
Evaluation Tools 68
Computing Power Related 69
Memory Related 72
Overview 72
V. **CONCLUSIONS & FUTURE WORK 74**
VI. **REFERENCES 77**

Index of Tables

Table 1. Cloud generation noises 20
Table 2. Reference hardware 33
Table 3. Vertical slice initial tasks 36
Table 4. Alpha initial tasks 37
Table 5. Beta initial tasks 37
Table 6. Gold initial tasks 37
Table 7. Initial cost analysis 39
Table 8. SWOT analysis 40
Table 9. Optimizations performance 73

Index of Figures

Figure 1. Battlefield 3 campaign level skybox 16
Figure 2. Billboard normal clouds in Sapiens 17
Figure 3. Billboard normal clouds (Schneider, 2015) 17
Figure 4. Billboard clouds in Sapiens 18
Figure 5. Polygon cloud (Schneider, 2015) 18
Figure 6. Geometry clouds in Minecraft 19
Figure 7. Absorption (Schneider, 2016) 21
Figure 8. In-Scattering (Schneider, 2016) 21
Figure 9. Out-Scattering (Schneider, 2016) 21
Figure 10. Silver Lining effect (Photograph) 22
Figure 11. Cloud dark edges (Schneider, 2015) (Photograph) 22
Figure 12.Horizon Zero Dawn clouds 23
Figure 13. Red Dead Redemption 2 clouds 24
Figure 14. Horizon Zero Dawn cloud gradient 24
Figure 15. Unity HDRP volumetric clouds 25
Figure 16. Sky Master ULTIMATE Unity package tool 27
Figure 17. Weather Maker Unity package tool 27
Figure 18. UniStorm Unity package tool 28
Figure 19. Unreal volumetric clouds 29
Figure 20. Unreal cloud material 29
Figure 21. Gantt project 30
Figure 22. Kanban project 31
Figure 23. Worley cell selection. 43
Figure 24. Worley 1 octave 44
Figure 25. 2D Improved Perlin vectors 46
Figure 26. Fade interpolation function 46
Figure 27. Improved Perlin 1 octave 47
Figure 28. FBM code example 47
Figure 29. Improved Perlin octave comparison 48
Figure 30. Remap function code 49
Figure 31. Cloud base texture channels 50
Figure 32. Cloud detail texture channels 50

Figure 33. Weather map channels 51
Figure 34. Initial Ray March sphere test 52
Figure 35. Early base cloud texture test 53
Figure 36. Density method code.. 54
Figure 37. Cloud generated with complex shape altering gradient 56
Figure 38. Complex shape altering gradient.. 56
Figure 39. Dual-lobe phase function 60
Figure 40. Henyey-Greenstein phase function 60
Figure 41. Light scattering initial approximation code 62
Figure 42. Light scattering integration 63
Figure 43. Light scattering integration code.. 63
Figure 44. Multiple scattering approximation total light contribution 64
Figure 45. Multiple scattering approximation octave light contribution........................ 64
Figure 46. Multiple scattering approximation octaves 64
Figure 47. Blue Noise comparison 66
Figure 48. Benchmark system graphic output.. 69
Figure 49. Dynamic Ray March steps with density (Schneider, 2015).......................... 70

Glossary

Abbreviations

E.g., *exempli gratia*, for instance, for example.

I.e., *id est*, that is, in other words.

Acronyms

CPU Central Processing Unit.

fBM Fractal Brownian Motion, fractional Brownian Motion.

GPU Graphics Processing Unit.

HDRP High Definition Render Pipeline.

HLSL High Level Shader Language

LUT Lookup Table.

PBR Physically Based Rendering.

RGB Red Green and Blue, usually image color channels.

RGBA Red Green Blue and Alpha (Transparency), usually image color channels.

UDP Universal Render Pipeline.

ND N-Dimensional.

(a) **2D** Two-Dimensional.

(b) **3D** Three-Dimensional.

Vocabulary

Algorithm (a) A set of mathematical instructions or rules that, especially if given to a computer, will help to calculate an answer to a problem. (b) A step-by-step procedure for solving a problem or accomplishing some end. E.g., a recipe is an algorithm, which consists of specific instructions for preparing a dish or meal.

Pseudorandom (of a number, a sequence of numbers, or any digital data) Satisfying one or more statistical tests for randomness but produced by a definite mathematical procedure.

Shader A user-defined program designed to run on some stage of a GPU. Shaders provide the code for certain programmable stages of the rendering pipeline. They can also be used in a slightly more limited form for general, on-GPU computation.

Volumetric Showing or creating something in three dimensions; e.g., a technique known as volumetric display creates moving 3D images that viewers can see from any angle.

Voxel Any of the discrete elements comprising a three-dimensional entity. The 3D equivalent of a pixel. E.g., an image produced by magnetic resonance imaging.

© Oscar Pérez Martín 2022

Preface

Motivation

As a programmer and game developer I love converting my thoughts, the inner worlds that I dream about, into tangible and interactable spaces that people can walk through and explore. I find it incredible that through changes in ones and zeros these worlds can be recreated and simulated to a certain extent inside small machines that we call computers.

I had been creating spaces that made use of rasterization [1]of 3D geometry for a long time before writing this thesis but I wanted to embark on a new **challenge** and experiment with a less commonly used rendering technique in videogames: **Ray Marching**;

Another field that I have always been drawn to is that of **Procedural Generation**, which allows a seemingly infinite amount of combinations and is already being widely used in videogames, as it can improve the pace at which content is generated while saving resources.

The blend of Procedural Generation with Ray Marching as a challenge to create my own worlds combined with my love for clouds as these dynamic, epic, physics-driven phenomena that occur every day almost unnoticed by people while being common and internationally recognizable among humanity, has led me to try to recreate these structures for use in videogames.

[1] A technique used to render of 3D scenes, mostly used in real-time applications; Explained in detail following the link (https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation)

The Problem

Skies in outdoor environments in videogames usually suffer from a lack of importance on the development stage of games as it is very easy to replace the background with a **Skybox**, a method that has been used traditionally; This makes the skies boring, **repetitive and static,** as the same image is shown to the users every time they look up, which does not contribute to the overall player immersion.

A solution has been adopted to solve the monotonous nature of static skyboxes: **rotating skyboxes**; these are normal skyboxes but they rotate at a constant rate along an axis to produce a more dynamic feeling. If the game has a day-night cycle the image can be rotated along an axis contained in a horizontal plane to show both a day and a night hemisphere; one clear example of this is Minecraft. Skyboxes in commercial engines that use images of clouds usually revolve around a vertical axis as if the clouds were moving.

With the increase of GPU performance and the **general adoption of shaders** in the industry in recent years more complex solutions have become available for developers with shader knowledge. **Real-time** shaders which rely on **atmosphere scattering approximations** are now available in most commercial engines causing the skies to react to light almost identically to the real sky to the human eye and changing its color depending on the time of day and dust in the atmosphere. With this, skies are now dynamic, but almost featureless, as they lack something that humans perceive as inherent to them; Clouds.

While other solutions like 2D billboard clouds with normals that react to light or 3D geometry exist, by far the most interesting and close to reality **solution** to that problem is the concept of **volumetric clouds,** which most of the time comes in combination with volumetric atmospheres. Volumetric clouds allow for the user to not only see the clouds but traverse them as they are 3D volumes and do not have hard edges as opposed to clouds represented with geometry. They are **already present in some** triple-A open-world **videogames** such as Horizon Zero Dawn and Red Dead Redemption 2 and they are making their way into the newer versions of the two most used commercial engines: Unreal Engine and Unity, but are t**oo complex for small studios** and indie developers to implement on their own with their **limited resources**.

The most advanced implementations of volumetric clouds that we find in video games are already premade so they do not offer customization to the user and the **tools**

© Oscar Pérez Martín 2022

for generating their cloudscapes are **not accessible** to developers of other games. On the other hand, plugins and systems for commercially available engines are too simple or do not offer great usability as most of them are conceived only as tech demo projects. The few systems that are usable and well-produced cost money or target high-end render pipelines, which limit the amount of small studios and indie developers that can access them.

Goals

General Objectives. This thesis has three main goals:

- **Create a Tool:** Develop a public and accessible tool to author volumetric cloudscapes for Unity games in real-time and document its development.
- **Develop a Project:** Complete a project, plan, develop and close it with a professional quality.
- **Contribute Knowledge:** Provide knowledge to the industry for those interested in the project and its systems.

Specific Objectives. To accomplish these general goals the following specific objectives will be targeted:

- **Texture Generation:** Make a system that allows for generation of 3D seamless textures which can be used to generate clouds.
- **Cloud Generation and Rendering:** Make a cloud generation & rendering system for the Unity Built-In Render Pipeline, making the clouds interact with light as accurately as in real life.
- **Customize the Clouds:** Allow users to customize their cloudscapes from presets, parameters and textures to better fit their own environments.
- **Create and Publish the Tool:** Publish the Tool as an open and accessible Asset Store Package.
- **Document Performance:** Analyse and document the efficiency of the methods used to generate and render cloudscapes.
- **Create a Demo:** Create a small Demo application demonstrating the performance and main customizable parameters of the tool, while letting the user explore the resulting cloudscape.

Scope of the Project

The **target** of this tool is not big videogame development studios but rather small ones, with few employees and a small budget; **indie game studios or amateur developers** who develop games as a hobby are good examples of our target audience. That is the reason why, of the three predefined Unity render pipelines[2], we chose to create the tool for the Built-In Render Pipeline, as it is the most used pipeline among our target audience and the easiest to set up for them.

Inside these small studios, our tool is **aimed** at their **artists** who are in charge of **creating the environments**; it will allow them to create cloudscapes that integrate well with the visual characteristics and aesthetics of their games. The tool has been developed to be **user-friendly** and to require **no programming** skills.

Players are the ones who will benefit the most from this tool as the environments on low budget and indie games they play will have enhanced visuals and provide better immersion as a consequence. **Developer teams** will **benefit** from the tool as well, owing to the fact that they will be allowed to produce more realistic and dynamic environments with no time wasted developing the systems and no money spent on them, enabling developers to spend their resources on other critical aspects of their projects.

This thesis' aim is not to create an entirely new system from scratch but rather attempts to adapt existing technology and knowledge into a tool that can be used in a commercially available engine.

[2] There are three main render pipelines available in Unity: (a) High Definition Render Pipeline (HDRP), (b) Universal Render Pipeline (UDP), (c) Built-In Render Pipeline. More information available in the link (https://docs.unity3d.com/Manual/render-pipelines.html)

State of the Art

A study of different techniques being used nowadays to model and light clouds is shown in this section as well as tools currently available in commercial engines.

Cloud Representations

When trying to represent clouds in games in real-time numerous approaches have been developed over the years; we are going to focus on the most used approaches in the industry for this thesis.

Skybox

In this technique clouds are embedded into the background image. Lighting of the clouds is baked or painted beforehand so no lighting calculations happen during the game execution. More advanced skyboxes can handle dynamic lighting for clouds but this technique is usually used in small levels where the player spends little time, so there is no need for dynamic skyboxes or change of lighting or shape of the clouds. It is the most used technique in fast-paced shooters, an example of which can be seen below.

Figure 1. Battlefield 3 campaign level skybox

Although not computationally intensive, it should be taken into account that skyboxes can allocate a lot of memory depending on the resolution of the image.

Billboard

In 3D, billboards are image planes that orient themselves towards a certain direction, either one of the main coordinate axis or a direction defined by the developers, like the camera vector in order to have the billboard face the player. Clouds are simulated in an external application and rendered usually as a color image defining the normal vector of each pixel; this direction is used then to simulate lighting direction of the clouds. Usually more than one image is taken for each cloud as seen from different angles and substituted or put together as the player moves around the world.

Figure 2. Billboard normal clouds in Sapiens

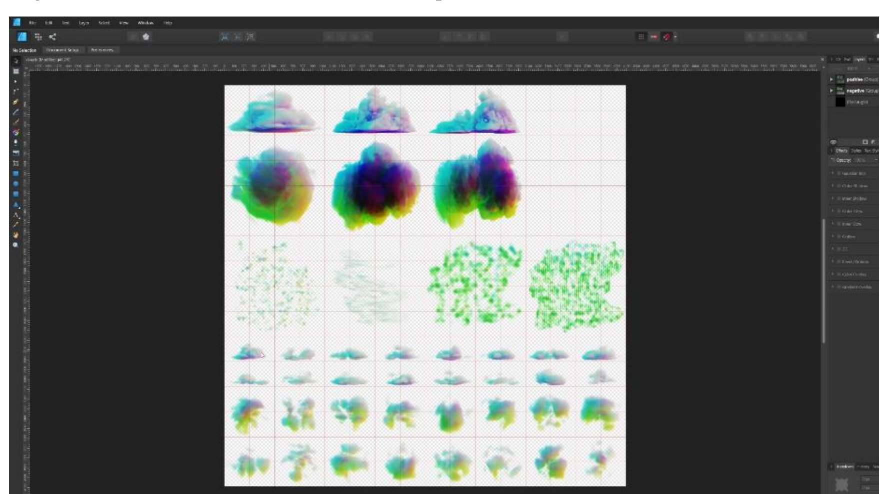
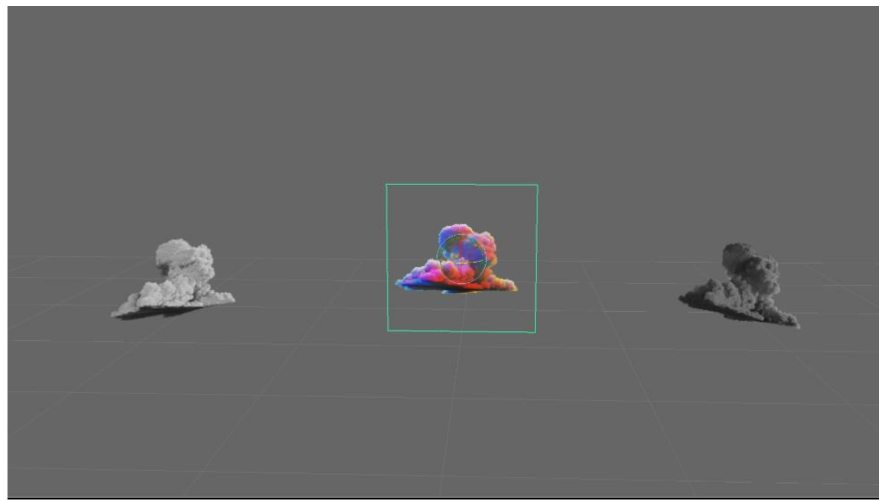


Figure 3. Billboard normal clouds (Schneider, 2015)



Billboard clouds offer a more interactable look than skyboxes as they are physically in a position in the world and can react to changes in lighting; one of the limitations of billboard clouds is that they are only suited to be viewed from afar since the illusion breaks once the player approaches or tries to fly through them. Furthermore, the shape of these clouds does not evolve over time and shadows cannot be casted between clouds realistically.

Figure 4. Billboard clouds in Sapiens

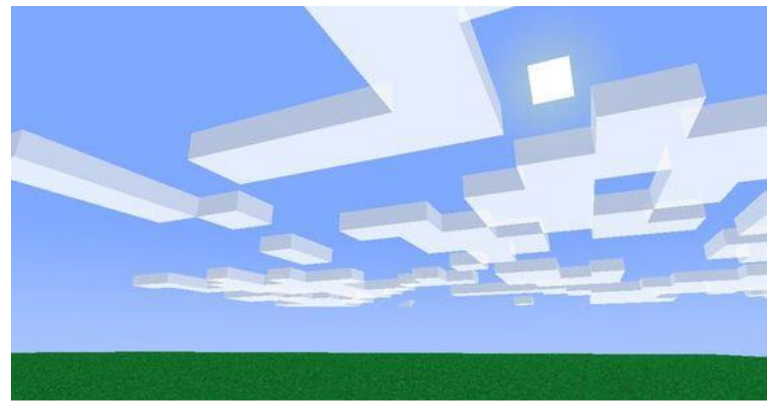
Polygon

With this technique clouds are constructed from vertices like normal geometry. They are created using data from simulations made in external applications and the lighting data is baked beforehand.

Figure 5. Polygon cloud (Schneider, 2015)

On the one hand, all game engines support vertex geometry so geometry clouds can easily be imported into different engines. On the other hand, their shape cannot evolve and clouds need have a high amount of polygons to look smooth enough to trick the human eye; In addition, they cannot be traversed as a volume without making the player see that they are only a shell. One example of a game that uses geometry clouds is Minecraft, although they are low-poly.

Figure 6. Geometry clouds in Minecraft



Voxel

Voxel clouds are comprised of a three dimensional grid of voxels or 3D pixels, each one storing a density value; Although they have to be created in an external application beforehand and their shape cannot evolve, they are volumetric in nature and can be flown trough. Voxel clouds use volumetric rendering, a technique detailed later. The major downside of Voxel clouds is their high memory usage.

Procedurally Generated Clouds

This is the technique that is being used in this thesis, it uses procedurally generated textures to define the shape and density of the clouds. It has very low memory usage, as it only requires textures, and can evolve over time. This technique uses the same rendering technique as Voxel clouds.

Procedurally Generated Content. Procedurally generated content is content that can be created from a set of rules or an algorithm, which saves memory. It is normally used in videogames to help developers quickly generate additional content and detail, saving them work; In this case it is used to generate noise textures.

Pseudorandom Tileable Noises. To generate volumetric clouds three types of noise are usually used:

Table 1. Cloud generation noises

Perlin	Worley	Perlin-Worley
8 octaves of Perlin noise	3 octaves of Worley noise	Worley noise being used to modify Perlin noise

The textures generated are usually 3D and tileable; i.e., when placing a texture next to the other, the change between textures cannot be noticed. The values stored in these textures are used to define the shape and detail of the clouds storing their density. Generating these textures procedurally allows the developers to generate different versions of noises only by changing the seed and a few parameters.

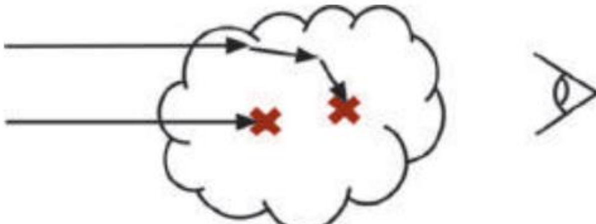
Volumetric Rendering

Volumetric rendering is a technique used to depict volumes which uses Ray Marching to sample the density of the clouds at points in space stepping along view rays from the camera.

It uses a lighting model derived from a simplification of how light interacts with clouds in the real world. It also uses Ray Marching to calculate the lighting towards the sun and describes three main ways in which the light can interact with particles in the cloud medium:

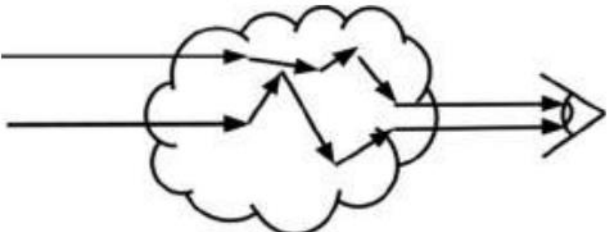
Absorption. The light ray can be absorbed by the particles inside the cloud. The further it travels inside the cloud the higher the probability for the ray to be absorbed.

Figure 7. Absorption (Schneider, 2016)



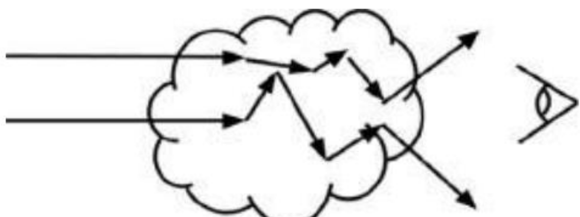
In-Scattering. The light ray can change course as a result of an interaction with particles inside the cloud and exit the cloud towards the eye.

Figure 8. In-Scattering (Schneider, 2016)



Out-Scattering. The light ray can change course as a result of an interaction with particles inside the cloud and exit the cloud traveling away from the eye.

Figure 9. Out-Scattering (Schneider, 2016)

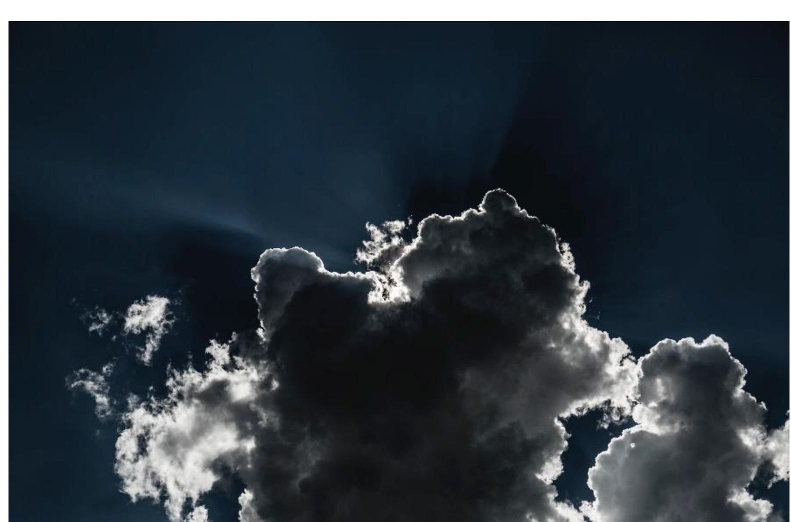


These light interactions can approximate two behaviours that occur in clouds in real life:

Directional Scattering. Gives clouds their luminous quality

Silver Lining. Highlights the edges of the clouds when looking towards the sun.

Figure 10. Silver Lining effect (Photograph)



However, there is a behaviour which they fail to approximate: the dark edges of the clouds when looking away from the sun.

Figure 11. Cloud dark edges (Schneider, 2015) (Photograph)

Different implementations try to solve this problem using various non-physical functions with distinct results.

Volumetric clouds with Ray Marching have only been seen in games in recent years due to the fact that rendering volumes is very computationally intensive, so the technology had to wait for powerful enough GPUs capable of performing this technique in real-time with shaders.

Volumetric Cloud Tools Available

Although there are very polished and capable tools in the industry, they happen to be part of big studios and are only used for a handful of games, which is the case with Nubis, one of the tools developed for Horizon Zero Dawn by Guerrilla, or the cloud system developed for Red Dead Redemption 2. These are capable of simulating a day/night cycle, smooth weather transitions and allow for a variety of artistic changes to better fit their game needs. They are also integrated into a greater sky and atmosphere system.

Figure 12.Horizon Zero Dawn clouds

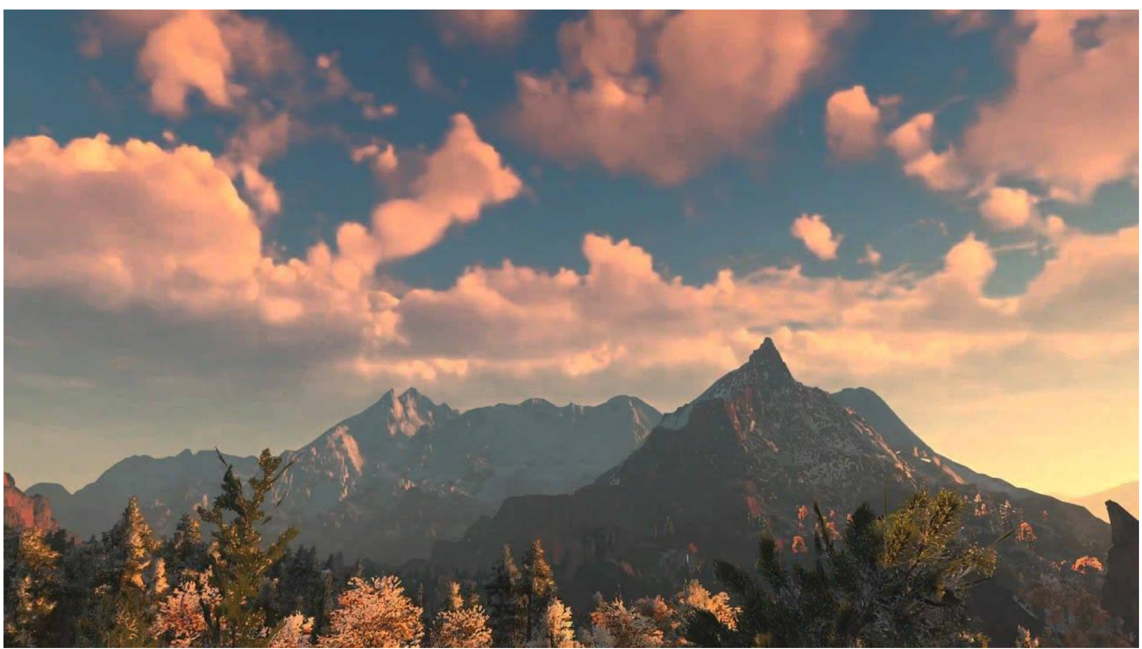


Figure 13. Red Dead Redemption 2 clouds

Horizon Zero Dawn's weather system has an interesting approach to making its clouds look more epic. A circular gradient is created around the player which tells the weather system to gradually transition to *cumulous* clouds at 50% coverage starting at a distance of 15 km. This makes sure that clouds at the horizon are always interesting and poke above mountains.

Figure 14. Horizon Zero Dawn cloud gradient

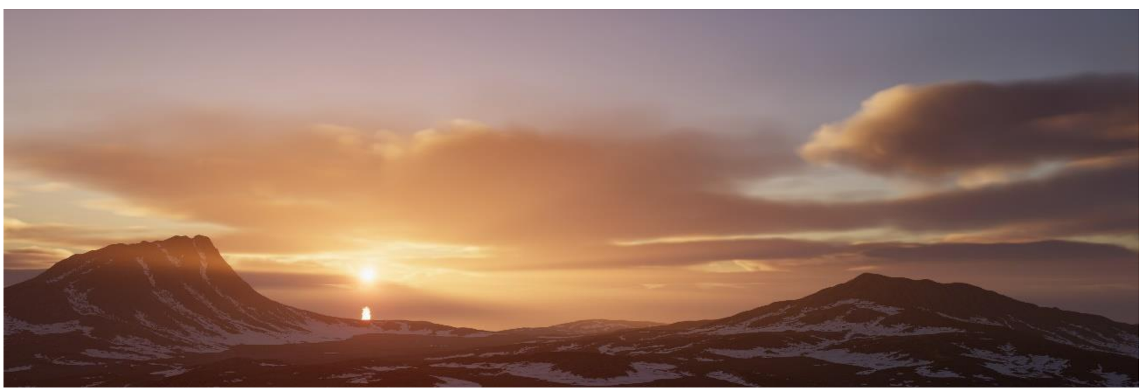
© Oscar Pérez Martín 2022

There are, however, free and accessible alternatives to these tools for the two biggest available commercial engines:

Unity Engine

Unity has recently released his volumetric clouds tool, a tool very similar to Nubis, very flexible and user-friendly; the only downside of the tool is that it is only available for the HDRP at the moment, leaving the majority of small studios and amateur developers that use less powerful render pipelines without the chance of using the system.

Figure 15. Unity HDRP volumetric clouds



This tool has three different modes allowing for three levels of customization, depending on the amount of control the developers want to have over the clouds:

Simple mode. This mode lets the user choose between four predefined presets: Sparse, Cloudy, Overcast and Stormy. Additionally, there is an option to create a customized preset with curves and parameters that can be changed to quickly create unique looking cloudscapes. Density curve by height, erosion curve by height, shape scale and offset and density multiplier are some of the settings that can be customized in this mode.

Advanced mode. Has similar options to the simple mode but with added settings to customize three different types of clouds: *cumulus, alto stratus* and *cumulonimbus*. These types of clouds and the rain regions of the world are controlled by a texture each that contains information about the cloud coverage for the former and rain distribution for the latter.

Manual mode. Shares settings with the previous modes but the cloud location and density, cloud type and rain location are controlled by a single texture which has all

that information encoded in its RGB channels. Furthermore, another texture serves as a LUT[3], encoding the following information in its color channels: cloud coverage (R), erosion (G), ambient occlusion (B).

Besides these modes, the tool offers earth curvature simulation, cloud layer height and thickness, wind skewing for cloud shapes, shadow casting on terrain, direct and ambient light color personalization and some quality controls to balance quality and performance.

Unity Packages

Unity has an asset store where users can submit their own tools and assets. There are good volumetric cloud tools for both HDRP and URP and some of them are also compatible with the Built-In Render Pipeline but most of them cost money so many people cannot access these tools. The tools available for free contain very basic functionality or are just demo projects with no personalization at all.

The following list contains the most relevant unity asset store tools that include volumetric clouds as a core part of their package.

Sky Master ULTIMATE. Includes volumetric clouds, lighting, PBR sky with atmospheric scattering, an ocean system and real-time global illumination. It has a sky manager that supports a day/night cycle and smooth weather transitions.

[3] Lookup Table. I.e., a predetermined array of numbers that provide a shortcut for a specific computation.

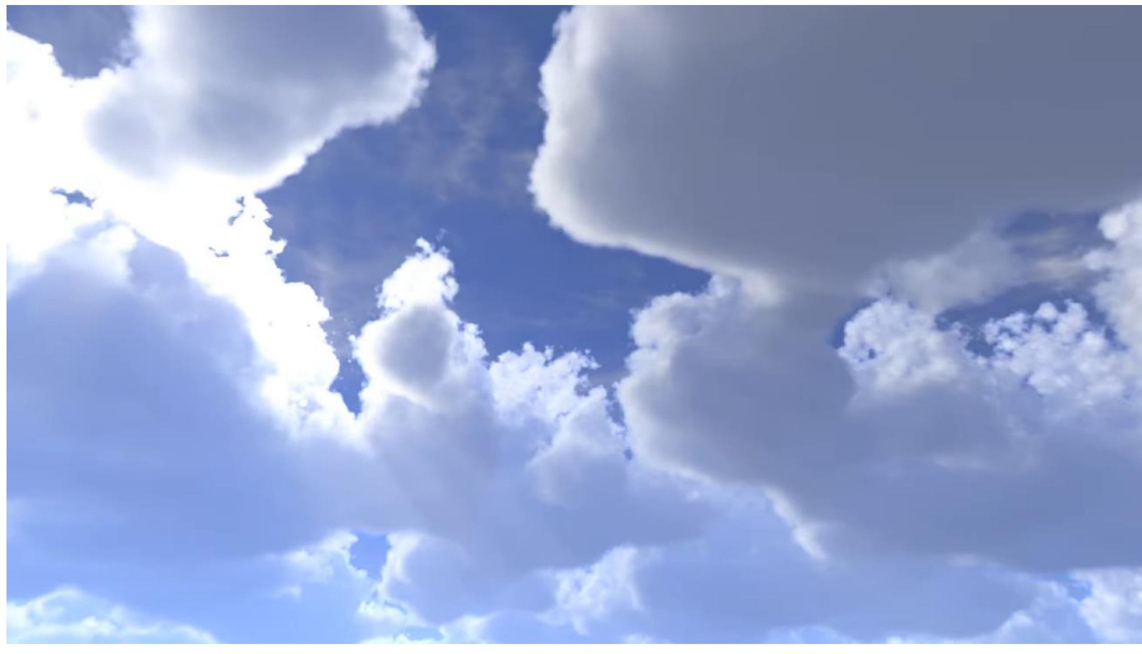
© Oscar Pérez Martín 2022

Figure 16. Sky Master ULTIMATE Unity package tool

This tool is compatible with both URP and HDRP Unity pipelines and its price in the asset store is 61.64€.

Weather Maker. Weather Maker supports (a) a day/night cycle; (b) volumetric clouds, fog and light; (c) terrain overlay; and (d) a sky system. Supports both 2D and 3D modes.

Figure 17. Weather Maker Unity package tool



This tool is compatible with both Built-In and URP Unity render pipelines and its price in the asset store is also 61.64€.

UniStorm. Includes atmospheric fog; cloud shadows; star constellations; customizable moon phases, cloud profiles and sounds for ambient and weather; procedural auroras; dynamic weather; and a day/night cycle.

Figure 18. UniStorm Unity package tool

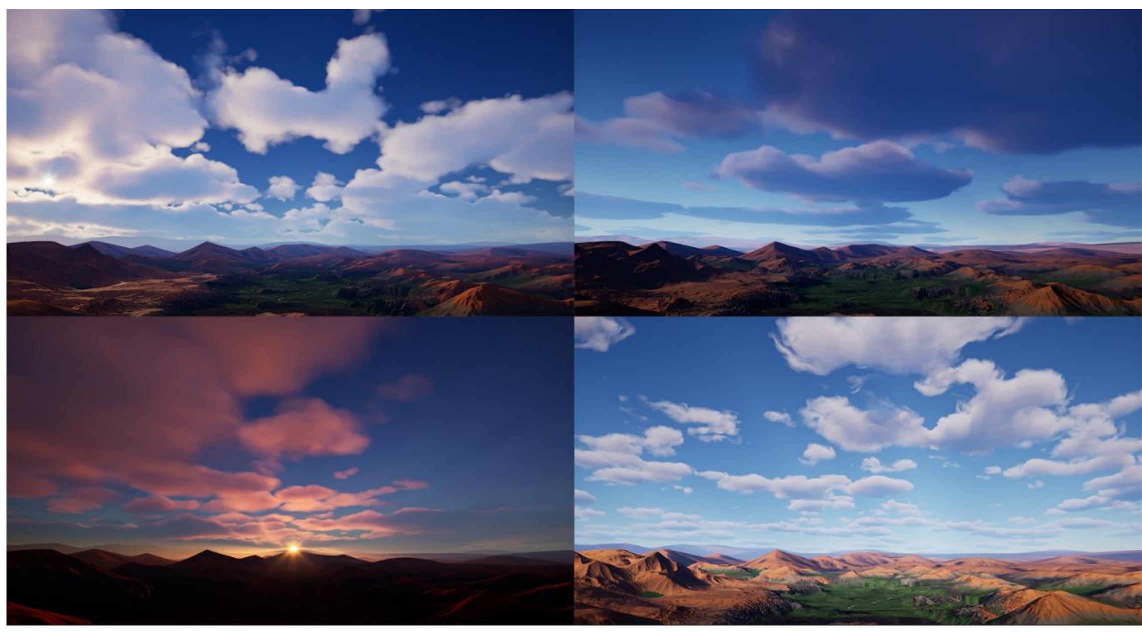
This tool is compatible with both Built-In and URP Unity render pipelines and its price in the asset store is 53.59€.

Unreal Engine

Unreal engine has a very powerful and easy to set up volumetric cloud system integrated with visuals similar to the Unity tool; however, it doesn't have that many options by default and it is less user friendly and more cumbersome to work with compared to its Unity counterpart.

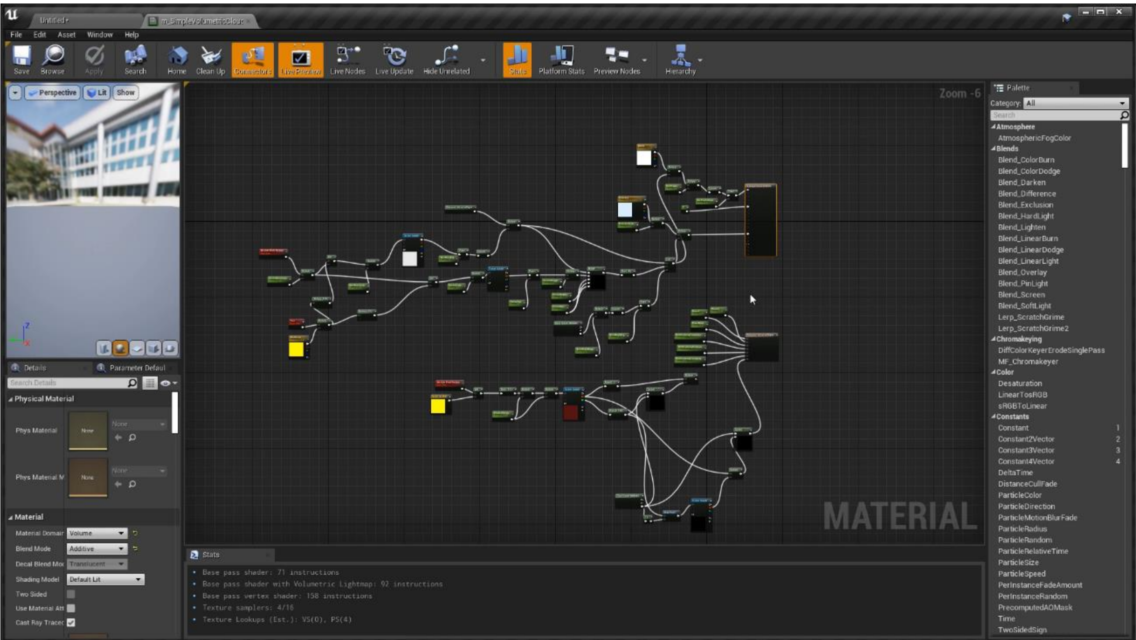
Cloud settings are scattered across the engine: the volumetric cloud object contains cloud layer related properties such as cloud height and thickness as well as atmosphere curvature; the main directional light contains light related settings such as transmittance or shadow extent; and the sky light object contains ambient occlusion related properties. This makes it difficult for the end user to adjust the clouds' look.

Figure 19. Unreal volumetric clouds



On the other hand, Unreal provides detailed and fine control for users that want to personalize every aspect of the visual quality of the clouds through a cloud material whose properties can be accessed through the material instance or changed in the material itself using a node based approach.

Figure 20. Unreal cloud material



Methodology

Documentation Structure

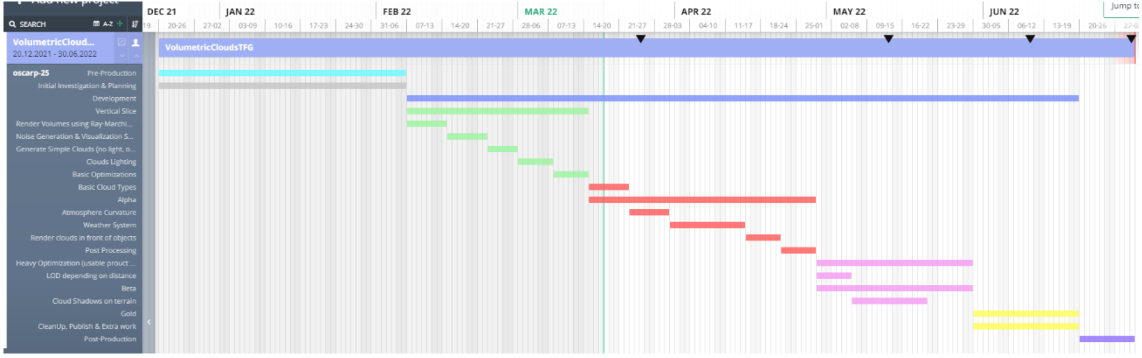
This thesis documentation follows a combined structure between the sixth and seventh editions of APA style[4], the thesis director's recommendations and the university's guidelines and thesis examples; although this is the norm, exceptions will be made since clarity of the thesis is prioritized. If a section of the document is hard to understand, the format and style will be adapted with the goal of making the thesis more readable and accessible to the reader.

Procedure and Tools for Project Monitoring

Gantt with Agantty

A Gantt chart has been created using Agantty. This chart provides us with a timeline with tasks that we can use to identify if we are ahead or behind schedule and adapt the project's pace and the tasks' priority and complexity accordingly. A detailed explanation of project phases and tasks can be found in the Planning section.

Figure 21. Gantt project



Kanban with Trello

To manage Gantt tasks with more granularity we use the Kanban method in a Trello board. We create smaller subtasks representing single features for the tool for each Gantt task which are managed through this task management process; the development

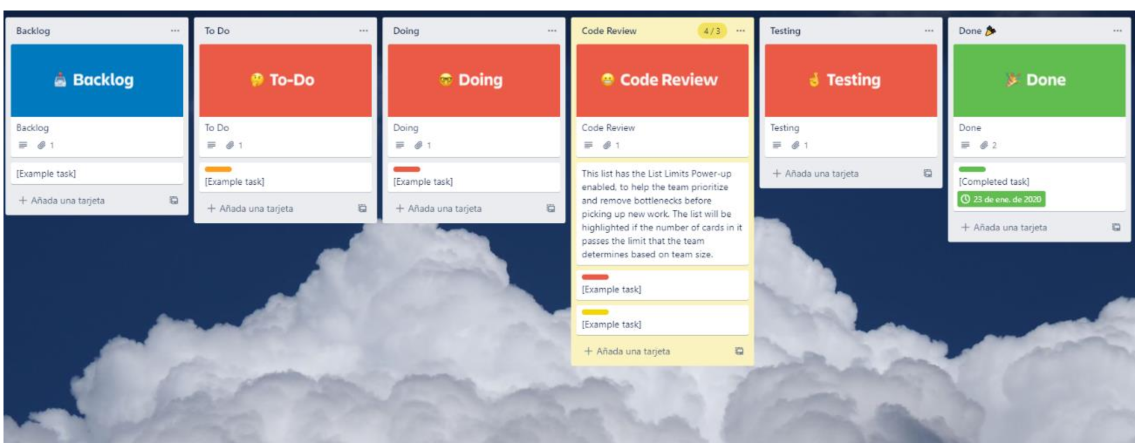
[4] American Psychological Association style (https://apastyle.apa.org/)

process of each task is divided into six different modules which the defined tasks have to go through in order to be completed.

1. **Backlog:** List of pending tasks.
2. **To Do:** List of tasks for the current Gantt task which are not being worked on yet but will be in the near future. Tasks are ordered by priority.
3. **Doing:** These are the tasks that are currently in progress.
4. **Code Review:** These are the tasks which need to be reviewed to check if they meet all the requirements needed to be tested.
5. **Testing:** Tasks which are currently being checked for bugs or errors.
6. **Done:** List of tasks which have already been reviewed and tested and are considered complete.

Tasks status and progress is evaluated and their scope is adjusted accordingly. Tasks have color tags which determine its priority: green (low priority), yellow (medium priority), orange (high priority), and red (critical, highest priority).

Figure 22. Kanban project



The *Backlog* section is used for tasks that will need to be performed in the future but should not or cannot be performed now or non-priority bugs that can also be solved in the future.

When a Gantt task needs to be started, it is divided into feature contained Kanban tasks which are placed under the *To Do* list. Tasks from the *To Do* list are being moved to the *Doing* list following two criteria:

1. Tasks with higher priority are performed first.

2. If the task is not the one with highest priority in the *To Do* list but a task with the highest priority in the *To Do* or *Doing* list depends on it, it is performed first.

To not be overwhelmed, the minimum amount of tasks possible will be in the *Doing* list at any given time.

Once a task is moved into the *Doing* phase the first thing that happens, if not done earlier, is extensive research on the task or the task feature. Once the possible implementations are understood or need to be tested to understand them, the task enters the *Development* phase in Unity. In this phase, if the task is complex, a first naïve implementation is done to quickly test its feasibility. If a task is determined non-feasible, more research or troubleshooting is done to conceive a viable alternative.

Once a task is deemed feasible a more thought-out implementation is performed. The implementation is iterated from a simpler version of the feature to a more complex and complete one; when the code implementation is finished and the task has the required functionality, the user interface in the custom editor is programmed, and then the task goes through a final polish pass. After that it is moved into *Code Review*. The *Testing* phase is detailed in the Task Validation section below.

During all the process of doing the task, if it exceeds its planned timeframe or it is considered not feasible to be developed in that amount of time, a revaluation of the task is performed according to the Risks and Contingency Plans section.

Version Control Tools with GitHub and GitHub Desktop

GitHub is being used to store all project files and code versions. New versions of the tool will also be uploaded to the GitHub repository during the project development using the GitHub Desktop app as commits. Demo application versions will also be uploaded to the GitHub Releases section; Builds will try to be made for capability demonstration purposes but are not strictly needed during development as the tool is intended to have functionality in the Unity editor and can be tested in real-time. At least one build will be created before finishing the gold phase and more will try to be made if tasks are being completed ahead of schedule.

The GitHub repository has two distinct branches:

- **Main:** Stable and feature-complete versions of the project are uploaded here.

- **Development:** In-progress versions of the project are uploaded here, they are not stable and the features in these versions may not be complete.

Evaluation Methods

Objectives Validation

There are two types of objectives that we need to differentiate when evaluating.

Subjective. The result of these objectives cannot be easily quantifiable. They need to be evaluated by a consensus between the thesis director and the tool developer. One example of an objective of this type can be if the rendering quality of the clouds is realistic enough.

Mesurable. These objectives can be measured and quantified. An example of such an objective is if the cloud rendering works in less than 16 milliseconds. To measure this we need to take into account the hardware used in the evaluation, in this case the reference hardware is detailed in the following table.

Table 2. Reference hardware

CPU	AMD Ryzen 5 5600X 6-Core Processor, 3701 MHz
GPU	NVIDIA GeForce RTX 2060
RAM	16,0 GB
System	Windows x64

Task Validation

Certain criteria must be met in order to consider a task as completed. A task enters the validation phase when it is moved from the Code Review to the Testing section on the Kanban Trello board. While the task is in this section, special attention is paid to it while the tool is running. Only one task can be in the Testing section at a time; if no major errors or bugs are found, the task is considered done. Minor bugs are considered a new independent task and the main task is still marked as completed.

Risks and Contingency Plans

Risks have been divided into two categories: General Risks and Concrete Risks; The former describes the most common risks applicable to any phase or task in the project while the latter describes the risks associated with certain tasks which are most likely to have problems. Not only the risks are described but also their solutions; Prevention work is also described for general risks.

General Risks

Bad planning. The task takes more time to be developed than initially planned.

Prevention. Project has been divided into phases to better acknowledge when there is a deviation from the planning with enough time to not affect the whole project. There is also a Clean Up task at the end of the planning to account for time deviations.

Solutions. Review features based on importance and delete or simplify some of the less important ones.

Too much complexity. A specific system or task is too complex, not fully understood or there is a lack of ability to correctly implement it. This can lead to trying different implementations which can lead to lack of time, in which case solutions for that risk apply.

Prevention. Previous research has been done so the level of complexity is already expected.

Solutions. Do some more research and/or ask the thesis director for advice. Solutions for the previous risk also apply here.

Concrete Risks

Clouds Lighting (T1.4). High Risk. There are different implementations possible and it is a complex topic. Can take a lot of time to get right. It is a critical feature of the project.

Solutions. Solutions are ordered from more to less preferred.

1. Take time from optimization or clean up tasks.
2. Being a critical feature it must be present in the project so the feature cannot be deleted, but can be simplified if necessary.

Noise Generation and Visualization System (T1.2). Medium Risk. The generation part of the system is complex and I have no prior experience programing 3D noises. Can take a lot of time to get right.

Solutions. Solutions are ordered from more to less preferred.

1. Use Unity built-in Perlin Noise to generate Improved Perlin Noise textures without the need to code them from scratch, saving time and having to program only Worley Noise.
2. Delete the Generation part of the system. Make users import pregenerated textures from external programs manually.

Planning

Phases of Development

This project development will be divided into three main phases similarly to videogames and software tool development.

Pre-Production

In this first phase all the research is performed to have a good understanding of the systems and features that we want to implement. Additionally, the planning of the project and its tasks is carried out.

Production

Also referred as development phase; in this phase the tool itself is created following an incremental and iterative agile model. The tool starts with basic functionality and more is added incrementally with each completed milestone; if necessary, tasks are changed and systems can be improved on in future milestones iteratively.

The development stage is comprised of four different phases or milestones, each of them divided into tasks that follow a Kanban agile methodology detailed in the Methodology section.

Vertical slice. Provides a demonstration of basic and minimum functionality of the tool.

Table 3. Vertical slice initial tasks

Task ID	Task	Start date	Due date
T1.1	Implement Ray Marching and render simple volumes to screen.	07/02/2022	14/02/2022
T1.2	Create a system to generate and display different types of noise in real time as tileable textures.	15/02/2022	22/02/2022
T1.3	Generate simple cloud shapes from noise textures.	23/02/2022	28/02/2022
T1.4	Create the lighting system for the clouds.	01/03/2022	07/03/2022
T1.5	Basic optimization.	08/03/2022	14/03/2022

Alpha. The tool is feature-complete at this stage, only missing UI/UX and some artistic elements.

Table 4. Alpha initial tasks

Task ID	Task	Start date	Due date
T2.1	Implement basic cloud types and cloud type map.	15/03/2022	22/03/2022
T2.2	Make atmosphere have a curvature.	23/03/2022	30/03/2022
T2.3	Implement a weather system.	31/03/2022	14/04/2022
T2.4	Render clouds in front of objects. I.e., Integrate clouds into the 3D world.	15/04/2022	21/04/2022
T2.5	Post processing. Atmospheric haze and light shafts.	22/04/2022	28/04/2022

Beta. During this stage, the main focus changes from adding features to integrating and optimizing the tool.

Table 5. Beta initial tasks

Task ID	Task	Start date	Due date
T3.1	More optimization. Make the tool usable for games.	29/04/2022	29/05/2022
T3.2	Level of detail for clouds depending on distance from the player.	29/04/2022	05/05/2022
T3.3	Cloud shadows on terrain.	06/05/2022	20/05/2022

Gold. In this last milestone the tool is finished, cleaned up and published, with the demo application also being finished.

Table 6. Gold initial tasks

Task ID	Task	Start date	Due date
T4.1	Clean up of the systems and code.	30/05/2022	19/06/2022
T4.2	Start the publishing procedure for the tool into the asset store.	30/05/2022	19/06/2022
T4.3	Prepare the demo project to showcase the tool.	30/05/2022	19/06/2022

Post-Production

Last phase of the project; in this phase, which happens once the tool is completed, bugs that have not been solved yet are continued to be fixed with the time left until the thesis is finished. Conclusions to acknowledge what has and has not work are elaborated.

Initial Cost Analysis

The project will be developed over the course of **five months**, with a dedication to development of **twenty-four hours a week**; this is relevant and is taken into account when calculating maintenance costs such as water or electricity. This project is not intended to make any profit. Costs have been divided into the following categories:

- **Personal and maintenance:** Monthly payments calculated from a salary estimation.
- **Software licenses:** Licenses for the tools used to develop both the project and the thesis.
- **Hardware:** Physical parts and electronic components needed to research and develop the thesis and the tool.
- **Books:** Books needed for research purposes.
- **Videogames:** Videogames needed for research purposes, which use systems related to the thesis.

Table 7. Initial cost analysis

Concept	Cost (€) Month	Cost (€) Total
Personal and maintenance	1332,55€	6662,75€
Salary	1290,10€	6450,50€
Water	7,15€	35,75€
Food	24,28€	121,40€
Electricity	11,02€	55,10€
Software Tools	-	0,00€
Agantty	-	0,00€
Trello	-	0,00€
GitHub	-	0,00€
Google Docs	-	0,00€
GIMP	-	0,00€
Hardware	-	1732.79€
Mouse	-	19,44€
Mouse Pad	-	12,75€
Keyboard	-	22,99€
Computer	-	1550,15€
Screen	-	146,90€
Books	-	141,14€
Production Volume Rendering Design and Implementation	-	51,23€
GPU Pro 7: Advanced Rendering Techniques	-	89,91€
Videogames	-	109,98€
Horizon Zero Dawn	-	49,99€
Read Dead Redemption 2	-	59,99€
Total		8646,66€

Salary makes up a large portion of the costs since the necessary materials are minimal and most of the software tools needed have free licenses or free alternatives.

SWOT Analysis

A SWOT[5] analysis will be used to assess our project's position compared with other tools and research papers in the field.

Table 8. SWOT analysis

SWOT	Positive	Negative
Internal	**Strengths** Worked with shaders previously, some experience in graphics programming. No great monetary cost or subscription software is needed for this project. Experienced working with Unity in different projects.	**Weaknesses** Not experienced in extensive academic research. First time developing a tool for Unity and publishing it as an asset store package. Inexperienced with Ray Marching algorithms and volume rendering.
External	**Opportunities** Other tools for Unity are far simpler, behind a paywall or not available for the Built-In Render Pipeline. A successful tool could be expanded and improved into a more complete one in the future, learning from the project problems and challenges. Extensive documentation for Unity and its Built-In Render Pipeline exist.	**Threats** Some tools for volumetric cloud generation already available for Unity. Unity volumetric cloud system adapted to the Built-In Render Pipeline could make the tool obsolete. Publishing the tool depends on Unity Asset Store approval.

[5] The SWOT (Strengths, Weaknesses, Opportunities, and Threats) analysis is a framework used to evaluate a company's competitive position and to develop strategic planning. SWOT analysis assesses internal and external factors, as well as current and future potential.

Planning Changes and Deviation (15/03/22)

Few changes have been made to the initial planning since the start of the project. One major change has occurred:

The cloud lighting task (T1.4) has been more complex to develop that initially thought. Different lighting methods have been experimented with to get the right look and performance for this system causing the task to take longer to complete than expected. This was anticipated in the Risks and Contingency Plans section being T1.4 a high risk task. This task is also critical; I.e., cannot be removed from the project.

The first solution for task T1.4 detailed in the Concrete Risks section has been followed; Task T1.5, basic optimization, has been removed as it has been assessed that at this point in time optimization is not yet needed. Following the solutions in the General Risks section, task T3.3, terrain shadows, has also been removed and task T2.5 has been simplified by removing light shafts from it. All tasks removed or simplified will be considered for future work. Removing or simplifying these tasks allows more time to be spent further developing task T1.4 until it is in a desirable state. No other task has been moved and task T1.4 will continue to be developed in the timeframe of the affected tasks to not further affect the project phases and timeframe.

Development

This section describes the development of the tool in Unity; all topics and systems will be explained in depth but in a way that the readers with little or no knowledge in programming will be able to have an overview of the development process.

Procedural Noise Generation

As mentioned in the State Of The Art section, the tool needs some textures to be able to generate the shape of the clouds as well as the weather map. To make the user not have to worry about creating the different textures necessary for it to function in an external program, to streamline the cloud generation process and to allow more variety in the shapes, I have opted to create a system that is capable of generating the textures needed within the unity editor in real time. This system is also designed to allow both the user who interacts with the tool and the developer to customize the textures and visualize the results in real time.

The first thing that needs to be done to start creating the noises is to have a way to debug them in the first place. This is done in a very simple fragment shader which, given the screen output texture and the texture we want to display, displays the latter in front of the former. The display texture can be scaled in the range [0, 1] being one the size of the shortest axis of the screen; We do that because the script that controls the shader has to work in a dynamic environment with changing screen dimensions as it is designed to work in Edit Mode; i.e., in a Unity mode where a script and its methods are executed in the editor instead of in the final game. This shader also has the capability of scaling down the coordinates of the display texture allowing users to control the amount of tiling of the texture that is seen on screen.

Two different types of procedurally generated noise are needed to create the textures that the tool will use and these noises are generated in a compute[6] shader to allow the generation to be computed fast and decoupled from the main rendering stage. As the noises will need to be tiled extensively, the implementations explained here have seamless borders and can be put next to themselves without any broken pattern or hard

[6] A compute shader is a Shader Stage that is used entirely for computing arbitrary information in the GPU. While it can do rendering, it is generally used for tasks not directly related to drawing triangles and pixels.

transition. The explanation for these implementations focuses on the 2D version of the algorithms with details on how to convert them to 3D but both the 2D and 3D versions have been implemented for the tool.

The texture display shader is used during the development process of the noises to help debug them better and faster in real time.

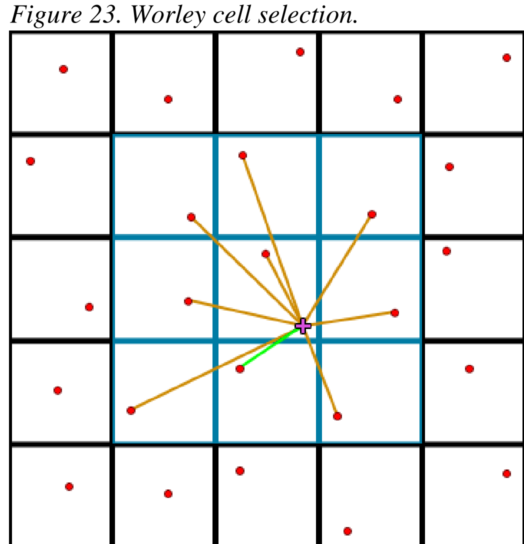
Worley Noise

Worley noise[7], also called cellular noise, is a type of noise based on distance fields; i.e., it represents the distance from the current pixel to the closest point in a set of points. This noise is often confused with Voronoi diagrams as they use this noise as a base.

While the implementation seems straightforward (iterate all the points in a loop for each pixel and find the nearest one calculating the distance between the point and the pixel), it becomes highly inefficient once we have a large set of points to evaluate. This, together with the fact that it makes it more difficult to create a seamless tileable noise, is the reason another approach is used. The space is divided in a grid pattern instead and only one point is placed inside each grid cell, in a pseudorandom location.

For each position that needs to be evaluated, the cell which corresponds to that position is located, alongside with its adjacent cells. Then its pseudorandom points are compared and the one closest to the desired position is selected. The distance between that point and the initial position is the generated value for that pixel.

Figure 23. Worley cell selection.



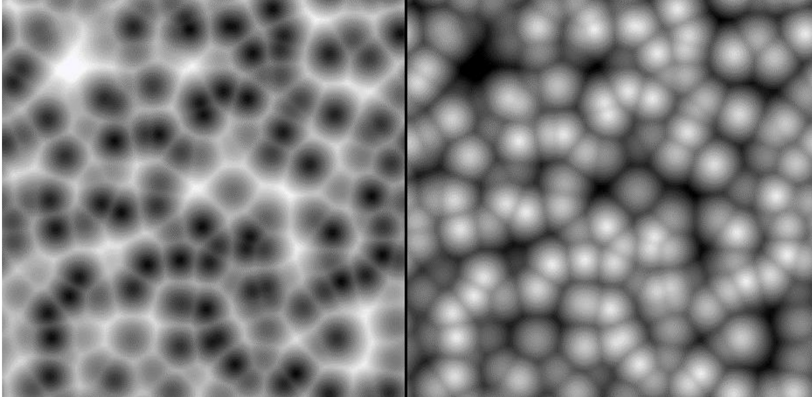
[7] An in-depth explanation and code implementation of Worley noise can be found in The Book of Shaders website, following this link: https://thebookofshaders.com/12/.

This approach gives a much more uniform look to the noise because of the cells of similar sizes and is easy to convert to a repeatable noise pattern simply by wrapping around the adjacent cells' positions when the position being evaluated is located in an edge cell in the texture.

All pseudorandom points generated for the cells need to be consistent no matter what position is being checked inside the cell. That means that for every pixel checked inside the cell the same point must be returned. Most of the implementations of the algorithm use a pseudorandom function to do this but another approach was used here instead, as it allows for more control: the pseudorandom points are generated in the script in CPU instead, and passed to the shader as an array of three-dimensional vectors. They are calculated once using Unity's own math library and their location can be controlled by a seed number.

The resulting image resembles a biological cell pattern (Figure 24.a) with bright edges, which is not what we want if we are trying to build billowy clouds, so a final step is required; the noise value has to be inverted from the [0,1] range to the [1,0] range highlighting the round shapes of the structure (Figure 24.b).

Figure 24. Worley 1 octave



The implemented noise is a 2D noise since it is easier to work with, but once it is verified that it works, the algorithm has to be converted to 3D so that it can be used in cloud modelling. With this noise the change is straightforward, instead of checking 8 adjacent cells we check 26 as we are in a three-dimensional space.

The texture display shader has to be modified to support displaying 3D textures; a variable with a [0, 1] range has been created to let the user control which slice of the 3D texture is displayed as a 2D texture.

Improved Perlin Noise

Improved Perlin noise is an improved version of the original Perlin noise by Ken Perlin. This improved version will be implemented as it gets rid of some directional artefacts and it is no more difficult to implement than the original while still being coherent noise with smooth changes.

Like the previous one, in this algorithm the space is divided into a grid of cells of equal size. This algorithm is divided into two different stages:

In the first one a pseudorandom value is generated from a position in space. This is done by a hash function that takes as inputs a permutation table and the current cell position. The permutation table is generated in the script outside the shader and consists of the values between 0 and 255 which are first shuffled. This permutation table is indexed using the X component of the position of the cell and the result is added to the Y component and is used as a new index of the permutation table. If the noise is 3D, the same procedure is done with the Z axis. Note that with this approach after 255 cells the noise will repeat due to the limited size of the permutation table. For us this is not a problem as the noise that is used doesn't have more than 100 cells in each axis. A problem occurs when implementing this: sometimes the index will overflow the permutation table size. Two solutions are proposed to solve this issue: The first one is to duplicate the permutation table after the shuffle, ending up with a 512 value table of two repeated sections. The second is to simply do the modulo operator of the index by the size of the table. The second solution has been chosen because of the memory savings when passing the table to the shader; the performance loss caused by the second option is not important here since the shader only runs once, not every frame.

The second part of the algorithm takes the pseudorandom number generated in the first part and uses it to index a small table of gradient vectors. In the 2D version of the algorithm the following eight 2D vectors have been used:

(1,1),(-1,1),(1,-1),(-1,-1),(1,0),(-1,0),(0,1),(0,-1)

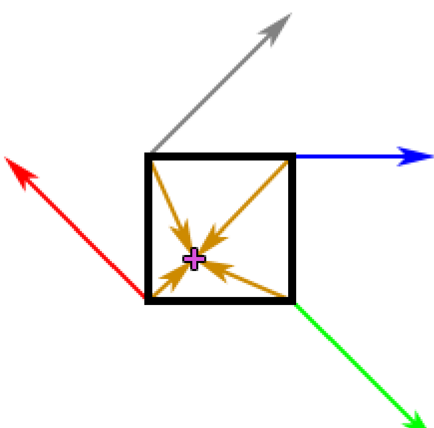
In the 3D version of the algorithm Perlin (Perlin, 2002) uses twelve 3D vectors, with the constraints that vectors must be skewed away from the coordinate axis and long diagonals to remove directional bias in the gradients; the same following vectors have been used to generate improved Perlin noise for the tool:

(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),(0,1,1),

(0,-1,1),(0,1,-1),(0,-1,-1)

When evaluating a position inside a cell, one gradient vector for each corner must be found. This is done by inputting the cell position to the algorithm to find the bottom-left gradient vector of the cell and the position of the next adjacent cells to find the gradients of the other corners. See RGBA color vectors in Figure 25 (Gradient Vectors).

Figure 25. 2D Improved Perlin vectors



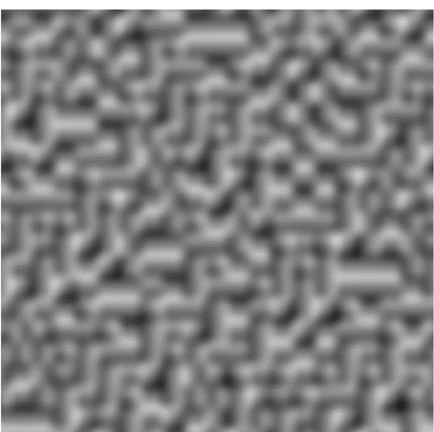
Another group of vectors is calculated by subtracting the point position from each of the corner positions. See brown vectors in Figure 25 (Position Vectors). For each corner the dot product between the gradient vector and the position vector of that corner is calculated. The resulting number is interpolated with the numbers of the other corners of the cell using (a) bilinear interpolation in 2D or (b) trilinear interpolation in 3D. The result of that operation is the value of the noise at that point in space; a value between the [-1, 1] range. Both the 2D and 3D algorithms have been implemented as they are needed for cloud erosion and the weather map. If linear interpolation is used it will result in abrupt transitions so a fade function provided in (Perlin, 2002) is used as the interpolation factor instead as seen in Figure 26.

Figure 26. Fade interpolation function

```
//Unoptimized version
function Fade(t){
        return 6*t*t*t*t*t - 15*t*t*t*t + 10*t*t*t;
}

//Optimized version (less multiplications)
function Fade(t){
        return ((6*t - 15)*t + 10)*t*t*t;
}
```

The resulting texture after the interpolation can be seen in Figure 27.

Figure 27. Improved Perlin 1 octave



Fractal Brownian Motion

Both Worley and Improved Perlin noises explained and generated in the sections prior are correct but lack detail and variety. One way of generating more detailed noises procedurally is using an iterative technique called fractal Brownian Motion (fBM from now on).

What fBM does is it adds up different textures with different intensities and varying dimensions. Three new variables (persistence, lacunarity and number of octaves) determine the look of the fBM noise as we can see in Figure 28:

Figure 28. FBM code example

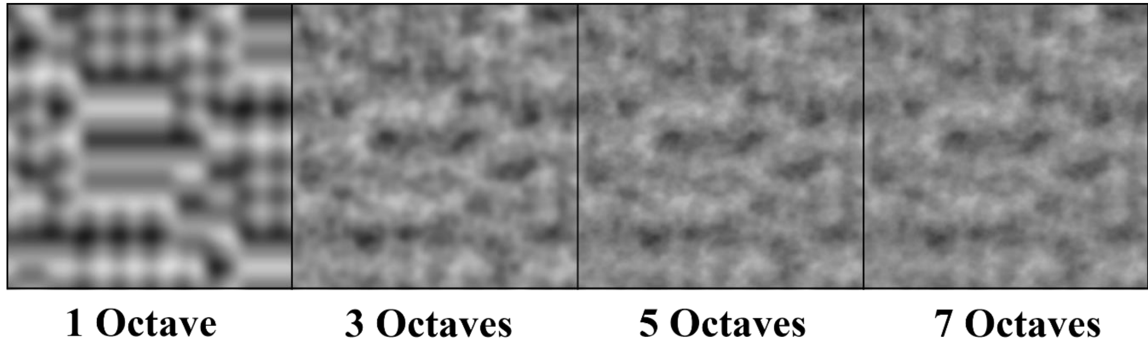
```
float result = 0.0;
float frequency = 1.0;
float amplitude = 1.0;
float totalAmplitude = 0.0;
for(int i =0; i<octaves; ++i)
{

    result += GetNoise(noisePosition*frequency)*amplitude;
    totalAmplitude += amplitude;

    amplitude *= persistence;
    frequency *= lacunarity;
}
result = result/totalAmplitude; //Normalize to -1,1
```

- The octaves determine the number of noise layers that need to be generated. The higher the number, the greater the detail, but also the greater the performance impact.
- Lacunarity is usually a number greater than one; it multiplies the frequency of the noise, making it noisier and more detailed with each octave.
- Persistence is usually a number less than one; it multiplies the amplitude and determines how much each octave affects the overall noise shape.

For Improved Perlin noise the fBM is implemented exactly as seen in Figure 28 to let the user customize all of the parameters, but for Worley noise the number of octaves has been set to 3 and for each octave the frequency (and thus the lacunarity) can be adjusted manually to give more control and achieve a more cloudy shape feel.

The maximum number of octaves allowed for the user in the tool for Improved Perlin is 10 but with more than 5 octaves the difference is negligible and not worth the cost as seen in Figure 29.

Figure 29. Improved Perlin octave comparison



Cloud textures

Following a similar approach to (Schneider, 2015), we will use two 3D textures to erode clouds in the density model:

The first texture has a default resolution of 128px * 128px but can be changed by the user to powers of two. It has four different channels (RGBA) and contains data used to create the base shape of the cloud:

1. R Channel: consists on a Perlin-Worley noise. A texture consisting of a customizable number of octaves of Perlin is mixed with a three-octave Worley noise. Usually to mix two textures, one would simply be multiplied by the

other but we want to erode the borders of the Worley noise with the Perlin noise to not have gaps in the denser parts of the texture. To do that a remap function is used; the remap function code can be seen in Figure 30. The Worley noise is passed as the first input in the function and the Perlin is passed inverted as the second one. The third, fourth and fifth inputs are -1,-1, 1 respectively. This channel will form the base shape of the clouds. These noises have low frequency and the shapes they create are big.

2. G Channel: Consists on a three-octave medium frequency Worley noise.
3. B Channel: Consists on a three-octave high frequency Worley noise.
4. A Channel: Consists on a three-octave higher frequency Worley noise.

Figure 30. Remap function code

```
float remap(float originalValue, float originalMin, float originalMax, float newMin, float newMax)
{
    return newMin + (((originalValue - originalMin) / (originalMax - originalMin)) * (newMax - newMin));
}
```

The second texture has a default resolution of 32px * 32px and can also be changed by the user. It should have three different channels (RGB) but due to engine limitations a four-channel texture has to be used (RGBA) where the alpha channel is ignored. It might be used in the future to encode more information. The three channels encode data used to create the detail shapes of the clouds:

1. R Channel: Consists on a three-octave low frequency Worley noise.
2. G Channel: Consists on a three-octave medium frequency Worley noise.
3. B Channel: Consists on a three-octave high frequency Worley noise.

The process used to generate the noises for these textures is modular, each channel is generated separately running the shader different times and using a Vector4 mask of ones and zeros; the data generated in the shader is written only to the channels with a one in the mask. While this is not the most efficient approach, it leads to a more modular and reusable code and, since it is not executed every frame, the performance of this shader is not a key factor to take into account.

© Oscar Pérez Martín 2022

Figure 31. Cloud base texture channels

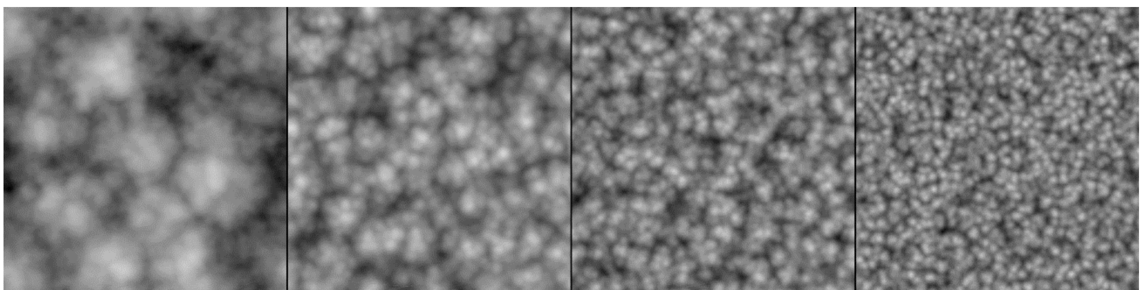
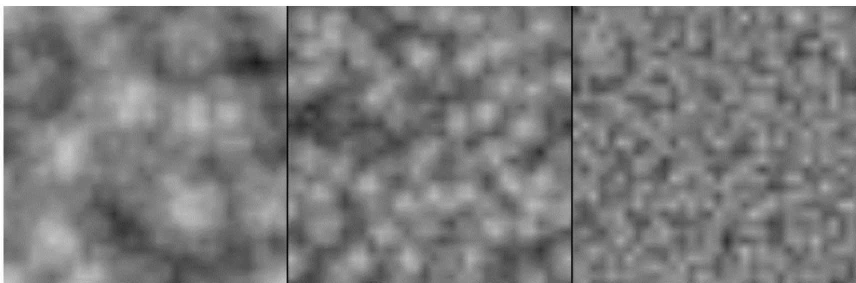


Figure 32. Cloud detail texture channels



Unity Scriptable Objects are used to store data regarding the noise settings. Scriptable objects allow both to modify the settings in real time and to save the variables as assets in disk to reuse them or swap them if needed. The texture generation script contains a list of noise settings Scriptable Objects for their textures and the correct ones are displayed in the custom editor of the script allowing them to be modified when their texture channel is selected.

Two types of noise settings exist in the form of Scriptable Objects:

- **WorleySettings.** Contains data to generate Worley noise: (a) the noise seed; (b) the frequency of the first, second and third octaves of the noise; and (c) the persistence value for the fBM.
- **ImprovedPerlinSettings.** Contains data to generate Improved Perlin noise: (a) the noise seed; (b) the number of octaves of the noise; (c) the persistence and lacunarity values for the fBM; (d) the frequency of the initial noise octave.

For the base and detail textures one WorleySettings Scriptable Object exist for each Worley noise channel and both a WorleySettings and an ImprovedPerlinSettings Scriptable Object exist for the Perlin-Worley channel.

Weather Map Textures

Initially the weather map texture was created in an external program but to give a more integrated experience in Unity and to achieve more consistent results the texture is now generated procedurally in a shader.

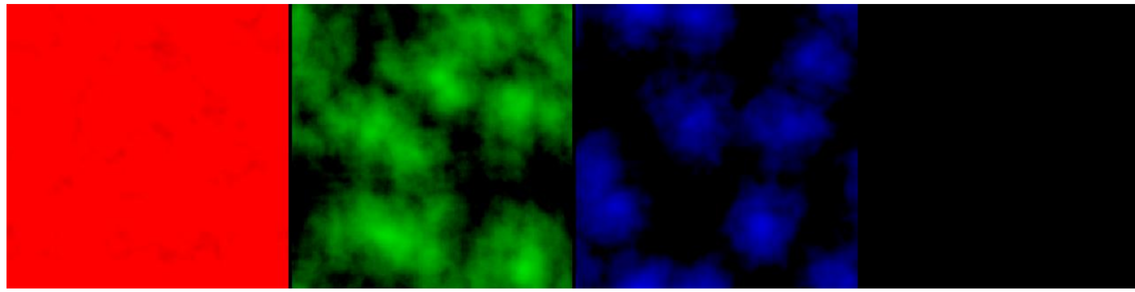
The first prototype texture is made of two different channels: a density channel (R) consisting of a Perlin-Worley noise and a cloud type channel (G) consisting of a Perlin noise.

This prototype weather map texture has been quickly discarded and updated to a better one which allows the placement of three different layers in the same XZ plane coordinates. This new weather map texture consist of three channels (RGB) containing the following data:

1. R Channel: encodes cloud layer 1 density.
2. G Channel: encodes cloud layer 2 density.
3. B Channel: encodes cloud layer 3 density.

It also contains a fourth unused channel that might be used for precipitation clouds in the future.

Figure 33. Weather map channels



Weather map textures also use channel masks and a modular generation process equal to the one used for the cloud textures.

Cloud Modelling

Ray March

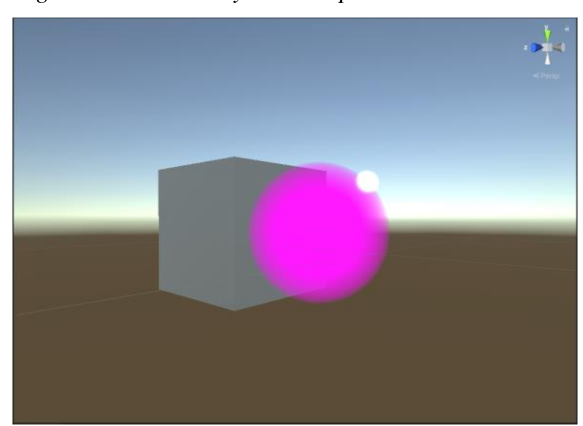
To properly model the cloud density and shape, the 3D space has to be sampled. This is done using Ray Marching, a technique that shoots a ray for each pixel in the

© Oscar Pérez Martín 2022

camera matching its perspective in the direction of the camera view and marches through the ray evaluating points in the ray direction every certain distance.

Initial tests have been done with a sphere intersection method, increasing a density value when the currently evaluated point in the ray is inside the sphere. This is a very simple test but allows to check that the Ray March matches the camera settings and perspective. The density value outputted has been used to then linearly interpolate between the color assigned to the volume and the scene view putting the sphere in a layer in front of what the player sees in the scene. A maximum number of points along the ray have been set to be tested so that the ray does not continue to evaluate points to infinity.

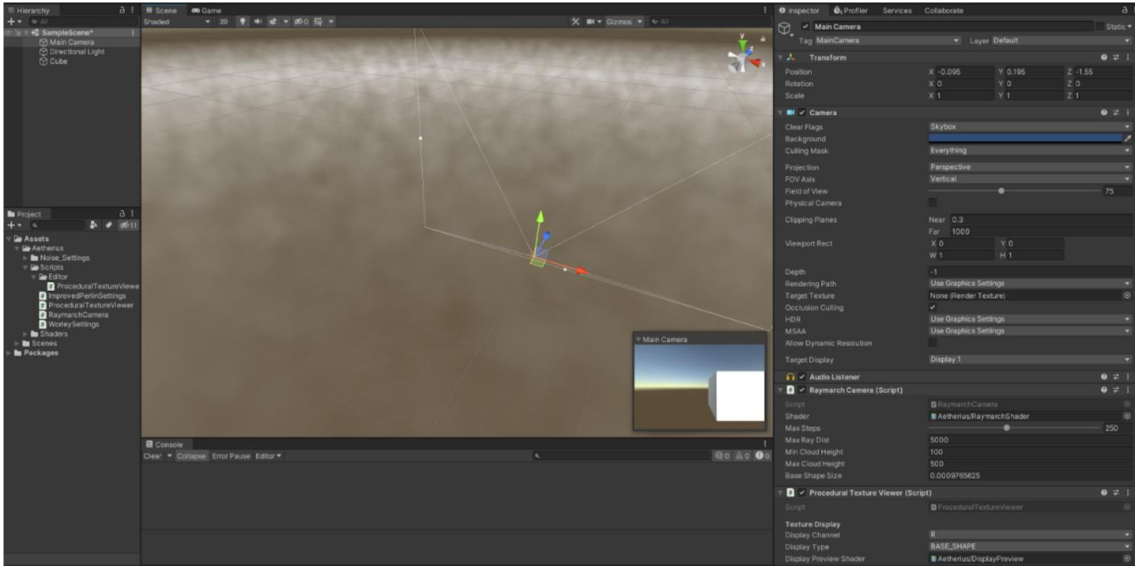
Figure 34. Initial Ray March sphere test



After the initial tests, clouds have started to be modelled. The first step is to determine the cloud layer extent; two variables determine the minimum and maximum cloud height and the cloud density is only sampled when the point currently being evaluated is located between these height values.

To test that the cloud base texture is correctly passed to the shader as a 3D texture, it is sampled from the evaluation position when it is inside the cloud layer resulting in Figure 35.

Figure 35. Early base cloud texture test



Density Model

The base cloud noise texture sampling described in the last section has been moved into its own GetDensity method which returns the density value for a given vector of 3D space coordinates. This is the method responsible for the cloud formations, their density and shape.

Following a similar approach to (Schneider, 2017) and (Häggström, 2018) the different input textures described in the procedural noise generation section are combined as seen in Figure 36.

First, all textures (base shape, detail shape and weather map) are sampled at the location given and stored into variables. The G, B and A channels of the base shape texture containing 3 different frequencies of Worley fBM are combined with a weighted sum to form an fBM of different Worley noise frequencies. The sum is executed as follows: (G Channel*0.625) + (B Channel*0.5) + (A Channel*0.125); the result is then saved in a variable which will be called *lowFreqFBM* for simplicity.

The cloud base noise now combines the main shape stored in the R channel of the first texture with this *lowFreqFBM* value. This could be done by multiplying them but we want to carve the noise into the base shape; following the same procedure as when Perlin-Worley noise was generated in the Procedural Noise Generation section, the remap function will be used.

The inputs for the remap function are the following: (a) base shape noise R channel; (b) negative inverted *lowFreqFBM* value, simplified to *lowFreqFBM* - 1; (c) 1; (d) 0; (e) 1. What this does is to take the base shape noise R channel value from the range [*lowFreqFBM* - 1, 1] to the range [0, 1], carving into it.

Figure 36. Density method code

```
float3 initialPos = currPos;
float density = 0.0;
float cloudHeightPercent = GetCloudLayerHeightSphere(currPos);//value between 0 & 1 showing where we are in the cloud

float4 weatherMapCloud = weatherMapTexture.Sample(samplerweatherMapTexture, (currPos / weatherMapSize) );
float4 lowFreqNoise = baseShapeTexture.Sample(samplerbaseShapeTexture, (currPos / baseShapeSize));
float4 highFreqNoise = detailTexture.Sample(samplerdetailTexture, (currPos / detailSize) );

//Cloud Base shape
float lowFreqFBM = (lowFreqNoise.g * 0.625) + (lowFreqNoise.b * 0.25) + (lowFreqNoise.a * 0.125);
float cloudNoiseBase = (Remap(lowFreqNoise.r, lowFreqFBM - 1.0, 1.0, 0.0, 1.0));
cloudNoiseBase *= ShapeAltering(cloudHeightPercent, weatherMapCloud.g, length(initialPos.xz - _WorldSpaceCameraPos.xz));

//Coverage
float cloudCoverage = weatherMapCloud.r;
float baseCloudWithCoverage = (Remap(cloudNoiseBase, 1.0 - globalCoverage * cloudCoverage, 1.0, 0.0, 1.0));

baseCloudWithCoverage *= DensityAltering(cloudHeightPercent, cloudCoverage);


////Detail Shape
float highFreqFBM = (highFreqNoise.r * 0.625) + (highFreqNoise.g * 0.25) + (highFreqNoise.b * 0.125);
float detailNoise = lerp(highFreqFBM, 1.0 - highFreqFBM, saturate(cloudHeightPercent * 5.0));
detailNoise *= 0.35 * exp(-globalCoverage * 0.75);

//Detail - Base Shape
float finalCloud = saturate(Remap(baseCloudWithCoverage, detailNoise, 1.0, 0.0, 1.0));

density = finalCloud * globalDensity;


return density;
```

The result is stored in a variable called *cloudNoiseBase* which outputs a homogeneous layer of cloud noise, now with more detail, but lacking the large clumps of dense noise that form the cloud shapes and the large voids where no clouds are present. To solve this the weather map texture density values are used. Clouds will form where more density is present in the weather map and no clouds will appear if the weather map shows no density. Another remap function is used to erode the weather map cloud formations with the billowy shapes of the base shape texture that the Perlin-Worley noise provided. The first input corresponds to the *cloudNoiseBase* value and the second input corresponds to the inverted weather map value times the coverage factor. The *cloudNoiseBase* value has first been multiplied by a shape altering method that given the height percent in the cloud layer returns a multiplier of the density at that height.

The ShapeAltering method contains two remap functions multiplied together, each of them generate a gradient. The former generates a gradient from black to white as a function of height and the latter a gradient from white to black. When multiplied

together they form a gradient which starts with no density at the bottom, reaches full density somewhere in the middle of the cloud layer and fades to zero density again at the top, generating what would be a vertical slice of the cloud shape density.

The result after eroding the *cloudNoiseBase* value times the ShapeAltering method by the weather map is a low resolution cloud shape that resembles real clouds but lacks some detail formations. This is fixed by using the sampled value of the detail shape texture to add more detail to the cloud.

Before adding the detail, the cloud shape is multiplied by a density altering method. This method generates a similar gradient to the one generated by the ShapeAltering method, the difference is that the latter defines the shape of a certain type of cloud and the former softens the top and bottom boundaries of the entire cloud layer to not have hard transitions.

To have more billowy details towards the top and wispier details towards the bottom, the detail value is interpolated by its negative version as a function of the height. This is then remapped into the main shape to carve out the details and the result is the final density value for a point in the cloud.

The key factors of the model are the following:

- Remap functions are used instead of multiplying values when we want to merge different textures to erode each other instead keeping the general shapes intact.
- Density inside the cloud changes with height depending on how close the evaluated point is from the cloud layer boundaries (DensityAltering method) and the type of cloud that is being generated (ShapeAltering method).
- Density of a vertical slice of cloud material is influenced by the weather map.

To sum up the model, the steps that need to be followed to create the cloud shape are the following:

1. Create billowy shapes eroding the base shape texture channel R with the other channels in the texture.
2. Modify this billowy shapes density as a function of the height with the ShapeAltering method.

3. Create the main cloud shape eroding the weather map, which is a vertically extruded version of the 2D weather map, by the billowy shapes.
4. Soften the cloud layer boundaries multiplying the main cloud shape by the DensityAltering method result.
5. Erode the main cloud shape by the detail texture, interpolated to cause wispy details at the bottom and billowy shapes at the top.

The clouds generated this way are very basic; more complex shapes are desired to make the tool more complete. This is the reason why the shape altering method has been modified to allow for more complex gradients. Instead of generating the gradients from two remap functions they are sampled from a curve defined by the user which represents a density value for each height percentage in the cloud layer. The curve is sampled into an array of values in the CPU when the curve is modified, and then passed into the shader where a density value from the array is chosen depending on the height percentage of the evaluated position in the cloud layer. More realistic shapes can be achieved with this method as seen in Figure 37:

Figure 37. Cloud generated with complex shape altering gradient

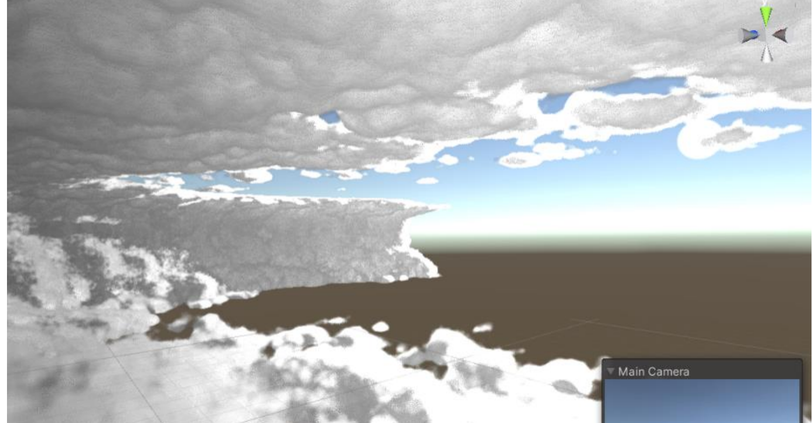
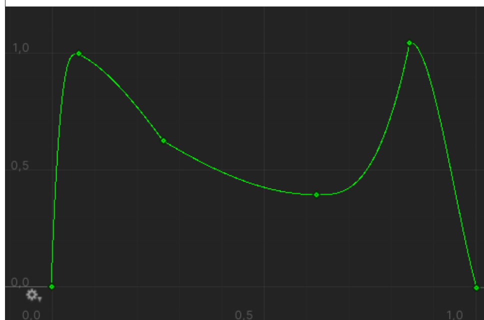


Figure 38. Complex shape altering gradient



Weather System

The weather system is a group of various features integrated inside the density model which makes the clouds feel more dynamic and interesting.

Wind and Skew

The first thing that has been implemented for this system is wind. Wind is simply an offset in the sample location of the base, detail and weather map textures. The shader receives a wind direction and the current game time and displaces the sample location accordingly. In practice, this makes the cloud noises and weather map scroll and move.

A multiplier for each texture sample has been added to let the user adjust how much the wind will affect each texture scroll. Furthermore, in some games the developers want the cloudscape to be in a certain way, they do not want the clouds to move but they still want the dynamism of moving clouds, so an option has been implemented for the wind to not affect the weather map. This way the player can perceive some movement in the clouds due to the different noises moving but the overall shape and location of the clouds in the sky does not change.

Skew has also been added to the clouds after the wind. This parameter distorts the overall shape of the clouds in the wind direction as if the wind was displacing them. Skew, like wind, woks by displacing the sample position of a texture, but it only displaces the weather map texture and the amount displaced increases with height.

Cloud Layers

To make the cloudscapes look more dramatic, a circular gradient like the one proposed in (Schneider, 2015) has been implemented and can be toggled on or off by the user. This gradient increments the presence of big clouds towards the horizon, making them more interesting. The distance from the camera at which this effect starts and the distance at which it maxes out can be controlled by the user of the tool.

Initially the weather system only supported one cloud layer with one cloud type which made the cloudscapes look very uninteresting and not realistic enough, as only one type of cloud could be displayed at once. That is why the system has been changed to support three different cloud types at the same time. The cloud type of a certain location on the weather map is determined by its green channel; A value of 0 means cloud type one, a value of 0.5 means cloud type two and a value of 1 means cloud type three. Any

value between zero and one linearly interpolates the ShapeAltering method explained in the density model for the two closest types of clouds. Although this new approach allows for more variety in the cloud formations, it still lacks realism.

Another solution has been proposed to solve this problem. Instead of encoding the cloud type in a weather map channel, three channels are used, one for each cloud type as detailed in the Procedural Noise Generation section. This allows for different cloud types, one on top of the other, effectively separating the cloud in three layers with independently customizable clouds for each layer. In the density model this is accomplished by performing all the steps for each cloud layer until the base shape for the clouds is created; then, the value with more density of the three layers is the one eroded by the detail noise.

This final 3 layer solution also makes it easier for the circular gradient towards the horizon to be implemented as it only has to affect the third channel value of the weather map texture.

Presets and Transitions

For the tool to be used quickly and easily with good results, some presets for different weathers have been created. These include the following weather presets: sparse, cloudy, stormy and overcast. When a preset is selected, a weather map for the preset is generated with predefined settings in code and passed to the Ray March shader. To not make abrupt changes in the cloudscape, the weather map also manages transitions between presets. When a new preset is selected the weather system starts linearly interpolating both the old and the new weather maps for a specified amount of time.

A problem identified with this approach is that if the user decides to change the preset when the weather system is already in the middle of a transition, the system will automatically finish the transition and an abrupt change will happen. There are three possible solutions to this problem:

1. Do not let the user change the preset when the system is in the middle of a transition. This solution does not involve a lot of work to be implemented but makes the transition instruction to be discarded and lost.
2. Queue the change and start the new transition once the last one finishes. This is a good solution but involves managing a queue and if a transition is queued it might happen later than expected by the user.

3. Bake the state of the transition into a texture at the time the new transition order is commanded and interpolate the newly generated weather map with this baked texture as the old weather map. This solution only involves a little bit of work and makes the transitions unnoticeable; this is also the solution chosen to be implemented.

To bake the texture a simple compute shader is created, which takes two textures and an interpolation value and generates the result of the linear interpolation.

Cloud Lighting

The cloud lighting model defines how light is propagated through the cloud medium. While both multi-scattering and single scattering models exist, the former is not suitable for real time applications, so a single scattering approach has been adopted. With a single scattering model, light is calculated for every point evaluated along the Ray March, casting a secondary ray towards the light source.

As described in the State Of The Art section the lighting in clouds is regulated by three phenomena which make the light exiting out of participating media be different of the light that has gone in:

- **Absorption**: the photons are absorbed by the medium matter, decreasing the light that reaches the camera.
- **In-scattering**: photons from all directions can change direction and scatter to the current light path, increasing the light that reaches the camera.
- **Out-Scattering**: photons that are traveling towards the camera are scattered away, decreasing the light that reaches the camera.

There is a fourth phenomenon which makes the media emit light when the temperature is really high. This effect will be ignored as we are working with clouds in a relatively cold environment.

The goal of the lightning model is to simulate those phenomena as accurately as possible. Two methods are going to be used to simulate absorption and scattering:

The Beer-Lambert Law is an extinction model (I.e., is concerned with how light energy attenuates over depth) that simulates light being absorbed when travelling through

a medium; a simplified version is used which takes as inputs the density accumulated in a certain distance and a term used to balance the solution. This is sometimes referred as the transmittance function: $T = e^{-d*t}$ where T is transmittance, d is density and t is the balancing term.

A phase function describes the probability distribution of light direction. Given an angle between the incoming light ray and the scattered light direction, the phase function tells how much light scatters towards this direction. The phase function is responsible for simulating both in-scatter and out-scatter events. The phase function of cloud participating media is very hard to model so the same approximation as in (Schneider, 2015) is used. The approximation used is the Henyey-Greenstein model, a phase function which can model directional scattering. This function has a problem: it is heavily biased towards one direction so when looking away from that bearing the clouds can appear nearly black as all the light is scattering towards the other direction. To solve this the approach presented in (Hillaire, 2016) is followed, which uses a dual-lobe phase function consisting of two Henyey-Greenstein functions blended together with a weight.

With this dual-lobe function both forward and backward scattering can be approximated.

Figure 40. Henyey-Greenstein phase function

Figure 39. Dual-lobe phase function

In the images, Figure 40 uses a *g* factor of 0.4 and Figure 39 uses a *g* factor of 0.4 for the first lobe and 0.5 for the second lobe with a blend factor of 0.2.

The first approach to lighting the clouds is a very naïve one; the initial system derives from early Ray March testing using density as a blend factor between a color and the background. That has become obsolete when lighting functions have started to be tested

and have not worked properly; a switch to an extinction model where work is only done to reduce the amount of light passing through the medium has been made.

As the cloud lighting system is very complex, lots of tests were made to try and define a realistic look for the clouds. That caused the task to take longer than expected and the appropriate actions detailed in the Planning section were taken in consequence. Some of the tests involve:

- Three different phase function implementations: (a) a simple Henyey-Greenstein function that only simulated forward scattering; (b) a Henyey-Greenstein function mixed with a secondary term that added more intensity around the sun for more dramatic sunsets; (c) the dual-lobe phase function detailed earlier.
- Three attenuation functions: (a) the simple Beer-Lambert Law simplification; (b) a Beer-Powder function used in (Schneider, 2015) to approximate the cloud dark edges when light scatters out of the cloud; (c) The attenuation function detailed in (Häggström, 2018), which uses the Beer-Lambert Law but with some light clamping.
- Additional non-physical alterations described in (Häggström, 2018) to (a) alter the dark parts of the clouds to appear brighter and (b) to approximate dark edges of the clouds using an out-scattering ambient function.

After the tests a lighting model mixing both the (Schneider, 2016) and (Häggström, 2018) approaches has been created. This model outputs the color of the cloud and the density and still mixes the background with the cloud color using this density. The light energy is calculated for every point in the Ray March process and multiplied by the transmittance result of the density at the current point times the length of the Ray March step.

To calculate the light energy, a secondary ray from the point currently being evaluated is thrown towards the sun direction and marches taking a few samples and accumulating the density values encountered. Transmittance is then calculated for this density along the light ray to account for absorption; this lets the clouds self-shadow. The result is then multiplied by an in/out-scattering method that uses a dual-lobe phase function mixed with a secondary term as described in the tests earlier which takes the angle between the light ray and the view ray as an input; this accounts for the silver lining

effect. The out-scattering ambient function from (Häggström, 2018) is used to approximate the cloud dark edges by multiplying it in the light energy calculation. Finally light energy is multiplied by the density and the step length, giving a more consistent result. The overall density is accumulated at each step and outputted to be used as an interpolation factor between the cloud color and the background color.

Figure 41. Light scattering initial approximation code

```
for (int currStep = 0; currStep < maxSteps; ++currStep)
{
    if (density < 1.0)
    {
        float currDensity = GetDensity(currPos);
        if (currDensity > 0.0)
        {
            float densityTowardsLight = DensityTowardsLight(currPos);


            float attenuation = Attenuation(densityTowardsLight,cosAngle);
            float inOutScattering = IOS(cosAngle,0.3,0.2);
            float ambientScatter = OSa(density, GetCloudLayerHeight(currPos.y, minCloudHeight, maxCloudHeight));


            lightEnergy += attenuation * inOutScattering * ambientScatter * currDensity * stepLength * transmittance;
            transmittance *= BeerLambertLaw(currDensity * stepLength, lightAbsorption);
            density += currDensity * stepLength;
        }
    }
    currPos += rd * stepLength;
}
density = saturate(density);
col = lightColor * lightEnergy * lightIntensity;

return float4(col, density);
```

While giving good visual results, two major problems have been encountered after implementing this approach which make it not ideal:

- Using the density as a mixing color factor with the background is not physically based and causes problems when the value exceeds 1, having to be clamped manually.
- Lots of Ray March samples are needed for the model to converge into a solution, usually more than 500, which decreases performance by a lot.

A complete rework of the lighting model has been done to solve those issues. The new integration model is energy conserving and needs an order of magnitude less of samples to converge; with less than 50 samples the results are accurate. An analytical integration is used to calculate the scattered light over a range described in (Hillaire, 2016).

The integration is defined as seen in Figure 42, where S represents the scattered light, σ_t represents the extinction coefficient, D is the integration depth and e is an exponential function. The exponential function is the Beer-Lambert Law simplification, the transmittance.

Figure 42. Light scattering integration

$$\int_0^D e^{-\sigma_t x} \times S \, dx = \frac{S - S \times e^{-\sigma_t D}}{\sigma_t}$$

When $\sigma_t = 0$ the result of the equation is undefined, so σ_t needs to be clamped to a small value. The extinction coefficient σ_t is calculated from the sum of two user tweakable values: the absorption coefficient σ_a and the scattering coefficient σ_s, being $\sigma_t = \sigma_a + \sigma_s$.

Figure 43. Light scattering integration code

```
float3 scatteredLuminance = float3(0.0, 0.0, 0.0);
float scatteredtransmittance = 1.0;
[loop] for (int currStep = 0; currStep < maxStepsRay; ++currStep)
{
        float currDensity = GetDensity(currPos,0.0);

        if (currDensity > 0.0)
        {
            float shadow = LightShadowTransmittance(currPos, 100.0f);
            float extinction = currDensity * extinctionC;
            float clampedExtinction = max(extinction, 0.0000001);
            float transmittance = exp(-clampedExtinction * stepLength);

            float3 l = lightColor * lightIntensity;

            float3 luminance = l * shadow * DoubleLobeScattering(cosAngle, 0.8, 0.5, 0.9) * currDensity;

            float3 integScatt = (luminance - luminance * transmittance) / clampedExtinction;

            scatteredLuminance += scatteredtransmittance * integScatt;
            scatteredtransmittance *= transmittance;
        }

    currPos += rd * stepLength;
}

return scatteredtransmittance * col + scatteredLuminance;
```

In the code example seen in Figure 43, S is represented by the luminance variable. The luminance here takes into account the light source color and intensity as the l variable, the transmittance in the secondary light ray as the shadow variable and the double-lobe phase function described earlier. Instead of outputting the density, the background color is multiplied by the transmittance and then the luminance which arrives to the camera is added to the result. There are, however, issues that have not been solved yet:

- No ambient light is taken into account when calculating light scattering.
- The dark edge effect of clouds is not simulated.

Multiple Scattering Approximation

A multiple scattering approximation has been used to better diffuse the lighting inside of the clouds as proposed in (Wrenninge, Kulla, & Lundqvist, 2013). It uses a summation over several scales to artificially lower the extinction coefficient σ_t, the phase function g factor and the scattering coefficient σ_s along the light ray, allowing more light to reach the sample point.

Figure 44. Multiple scattering approximation total light contribution

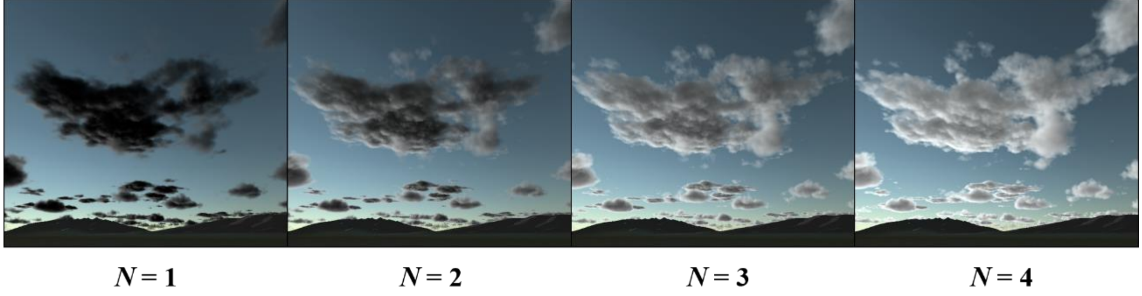
$$L = \sum_{i=0}^{N-1} L_i$$

Figure 45. Multiple scattering approximation octave light contribution

$$L_i = \sigma_s b^i L_{light}(\omega_i) p\left(\omega_i, \omega_o, c^i g_0\right) e^{-a^i \int_0^t \sigma_t(s)ds}$$

In Figure 44, N is the number of octaves. In Figure 45, a is the attenuation, b is the contribution, and c is the eccentricity attenuation. $N = 4$ is a good value for the tool although it can be changed by the user in the inspector. a, b and c are set to 0.3, 0.75 and 0.5 respectively and cannot be modified through the inspector.

Figure 46. Multiple scattering approximation octaves



$N = 1$ **$N = 2$** **$N = 3$** **$N = 4$**

Day / Night Cycle

To approximate the ambient light interaction between the skybox and the clouds, the following method has been implemented:

The ambient light directly above the player and the ambient light towards the sun have been retrieved from the ambient spherical harmonics probe. With a method to retrieve the luminosity from a color (0.299R + 0.587G + 0.114B), the luminosity value of both ambient colors is calculated and the maximum of the two is saved. This value

then multiplies the sun color, making it black when the ambient color is dark, at night. The dot product between the up direction and the sun direction is used to determine the sun inclination and increase the ambient color contribution towards the horizon, creating more epic dawns and dusks. On the shader side, the ambient color is remapped to the cloud height, being brighter the further up the evaluated point is in the cloud layer.

Scene Integration

In this section some features that aim to make the cloudscapes to be better integrated into the Unity 3D world are described.

Object Occlusion

Clouds need to interact with objects around them, being able to both occlude geometry, and be occluded by it at the same time, otherwise cloudscapes will be rendered in front of everything else.

Unity already renders a depth texture for each frame so it can be retrieved on the shader to compute occlusion. There are some problems that need to be solved when working with a Unity depth texture, however; the depth values in the texture are not linear but we want linear depth to calculate distances reliably. Fortunately, Unity has a built-in method in HLSL which converts depth to normalized linear values ranging from 0 to 1. These values can then be converted to meters using the far plane distance of the camera and passed to the Ray Marching method which will stop rendering clouds if they exceed the calculated distance.

A side effect of this feature is an increase in performance when the sky is blocked by objects as clouds do not need to be calculated.

Banding Reduction

One artefact which decreases the realism of the clouds is banding. This phenomenon is caused by the step size between the samples in the Ray March; as all the rays start at the same point and advance the same distance every frame, slices of the clouds are created at regular depths from the camera causing hard serrated edges to appear. To diminish the effects of banding, a simple solution has been adopted: make the rays start with slightly different offsets. The offset distance ranges from zero to one step.

Instead of using random values for each ray, a texture with blue noise is sampled to retrieve these offsets. The advantage of blue noise is that its values are evenly distributed; viewed from afar it appears as a featureless grey color texture.

Figure 47. Blue Noise comparison

Figure 47. (a) Some clouds with banding without the blue noise. (b) Blue noise active, no banding is perceived.

Atmosphere

In this section, atmosphere is defined as the region between the lower and upper boundaries of the cloud layer; I.e., the region where clouds can form.

In the initial naïve implementation of the atmosphere, the region is defined by two planes or distances from the ground plane. The purpose of this implementation is to have an early prototype to allow for quick tests of the clouds. Therefore, no raycast against the planes is performed to find the starting point and optimize samples, the Ray March algorithm only tests if its Y coordinate is between those distances and processes clouds accordingly.

This implementation has two drawbacks:

- Some computational power is used in checking points were there are no clouds.
- The clouds are parallel to the horizon, there is a gap between the lower bound of the cloud layer and the horizon. This is unrealistic as in real life the atmosphere is curved and clouds disappear behind the horizon.

To solve these problems a curved atmosphere has been implemented. The atmosphere lower and upper bounds are defined by two concentric spheres. A sphere-ray

intersection function is created, which given a ray returns the points at which the ray intersects the sphere.

A method has been created which takes the camera position as an input and returns a ray origin from where the Ray March starts and a maximum length for the ray. This method has three different behaviours depending on the camera position:

Below the atmosphere. If the camera is below the atmosphere, the method detects where the ray casted from the camera intersects the interior sphere in front of the camera and sets that as the starting position for the Ray March. It also tests the collision with the outer sphere to detect where it has to stop and returns the length between the start and end positions.

In the atmosphere. The method considers the camera position as the ray origin and the ray length is computed from the first positive intersection between any of the two concentric spheres.

Above the atmosphere. When the camera is above the atmosphere, the method first determines if there is an intersection with the outer sphere; if there is, it checks whether the ray intersects first with the inner sphere or a second time with the outer sphere and uses this intersection to calculate the ray length.

An issue has been encountered after implementing this method: when transitioning from being below the atmosphere to inside atmosphere, some clouds towards the horizon disappear. This is due to the ray stopping at the inner sphere intersection.

A solution has been implemented to fix this popping behaviour: when inside the atmosphere, the algorithm only checks for intersections with the outer sphere. This solution comes at the expense of larger Ray March steps and less optimized sample points when looking below the horizon as the ray travels to the other side of the atmosphere. It also renders all the clouds of the atmosphere, including the ones at the other side of the globe, which is computationally expensive. To minimize the drawbacks, a third sphere has been created simulating the ground of the planet; when a ray intersects this ground plane, it stops.

Custom Editor

Custom unity editors have been created for the scripts in the tool. This has been done mainly for two reasons:

1. A complex user interaction with the tool settings. There are lots of settings that users can customize in the tool, but they might feel overwhelmed if all of them are displayed at the same time. A custom editor helps by only showing the settings needed depending on the context.
2. Due to some scripts needing to be executed in the scene camera, methods that are called regularly can sometimes be called by that virtual camera, overwriting data and causing errors. With custom editors a workaround can be followed as the methods can be called from the editors themselves when a variable changes.

Optimization

Since the tool uses mostly procedural content, RAM usage is not an issue and does not need to be optimized. Instead, the focus is on performance optimizations and the metric used to assess performance is the time in milliseconds (ms from now on) it takes to render a frame or its inverse, frames per second (fps from now on). The goal is to render each frame in 16ms or less on the target hardware at full screen (1920px * 1080px).

The different optimizations have been divided in two different categories depending on what their target is: Computing Power Related and Memory Related. These categories are detailed below.

Evaluation Tools

Some tools have been used when trying to evaluate performance for the clouds, both external and Unity's own internal ones. These are the tools that have been utilized:

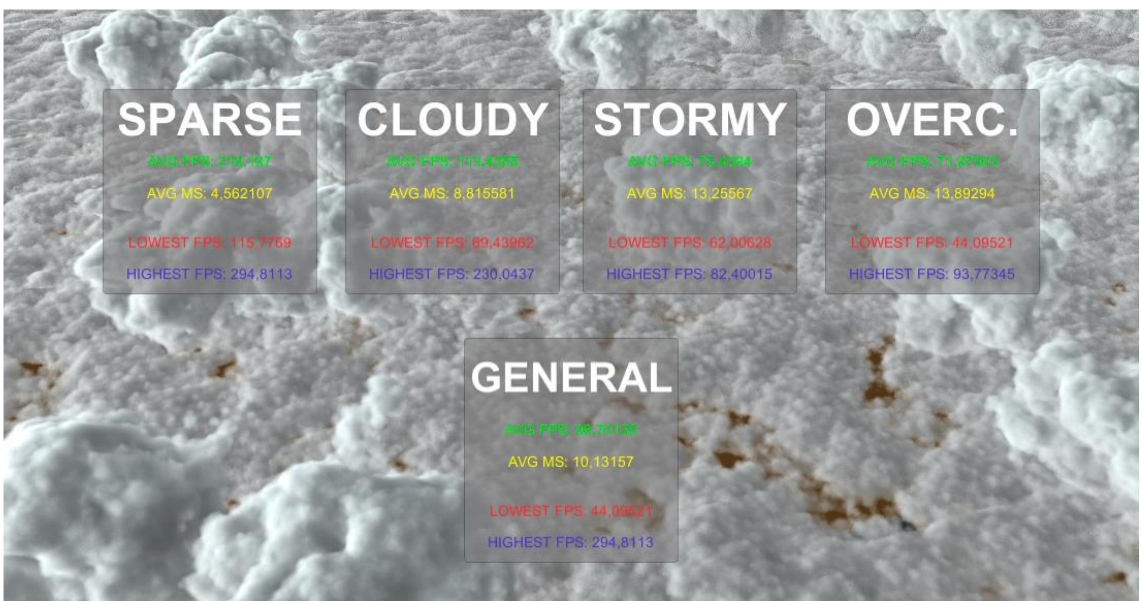
- **PIX:** An external tool for performance tuning and debugging for DirectX 12 games on Windows.
- **RenderDoc:** A graphics debugger tool that allows for single-frame capture and inspection of applications in a variety of platforms and graphics languages. It is available as a standalone application and is also integrated in Unity for easier debugging.

- **Unity's frame debugger:** An internal tool in Unity that outputs performance metrics for the overall frame and the functions that have been called in each of them. It does that for the last frames recorded in Unity's own editor.

Despite being useful in some specific cases in terms of finding errors, they are very cumbersome to use for fast iteration. The internal ones only work in the editor and Unity's own performance impact pollutes the metrics, and the external ones have to launch the standalone application that wants to be tested, collect the metrics, save them externally and the user still has to search for the important performance indicators on the data collected, which makes it very time consuming.

For these reasons a benchmark system has been created as a script in the tool. This system is part of the demonstration application and works both inside the Unity editor when play mode is active and in the standalone application. It only collects the data needed for the tool, it evaluates the performance at different altitudes in the sky in each of the weather presets available and outputs the information directly to the screen. This data can also be saved in a .csv file and opened with Excel or similar programs. All the metrics shown in the optimization section have been collected with the benchmark system.

Figure 48. Benchmark system graphic output



Computing Power Related

Optimizations explained here focus on reducing the time it takes for the shader to execute its operations, which is directly related to the number of operations needed to be

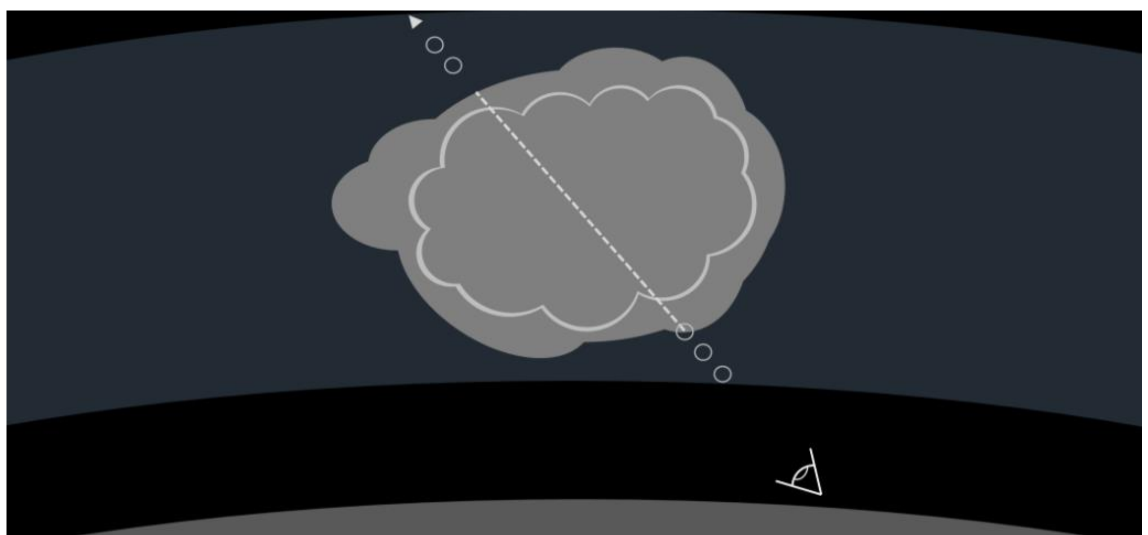
© Oscar Pérez Martín 2022

performed and the time cost of each type of operation. E.g., the GPU takes longer to compute a square root function than a simple multiplication of values.

Dynamic Steps with Density. Sampling textures on points along a ray is costly in terms of performance so we want to reduce the number of samples taken without a noticeable impact on the visual quality of the render. We can take advantage of the method that is used to get the density of the clouds at any point in space. As explained in the Cloud Modelling section, what this method does is to carve a base texture with other textures to get the final shape of the clouds so the shape is always contained within the base texture boundaries. This gives us the opportunity to divide the sampling method into two: the first is the detailed one, the one that has been used until this point, which is expensive and samples three textures; the second one is the cheap one, only sampling the base texture to know if the sample point is near a cloud.

As proposed in (Schneider, 2015), instead of using a constant step length during the Ray March process, the shader now uses a two-level approach; Points are sampled at a greater length with the cheap density method and once the density sampled is non-zero the shader is potentially sampling a point inside a cloud. It then goes a step back and starts sampling with the detailed density method at closer intervals as it did before the optimization. When a certain number of points in succession encounter no density in the detailed mode, the shader switches back to cheap samples and long steps. This is illustrated in Figure 49.

Figure 49. Dynamic Ray March steps with density (Schneider, 2015)



Dynamic Steps Over Distance. Since the nearest clouds are occupying most of the screen, they need a great amount of detail, whereas far away clouds with a span of only 10 pixels do not as it won't be visible. With the constant step over distance approach that was first implemented, the detail density did not change wasting samples on far away clouds and rendering them with the same detail as the nearest ones. With this optimization it is proposed to, after a certain amount of distance from the camera has been reached, switch from constant samples to dynamic ones, increasing the distance between samples the further away they are from the camera. This reduces the amount of samples needed to be evaluated for each ray and increases performance. The implementation used in the tool linearly interpolates between a minimum and a maximum step length given a value between zero and one. This value is calculated as the percent that is the sampled distance from the maximum possible length a ray can be in the spherical atmosphere squared.

LODs Over Distance. A very straightforward optimization already present in meshes for most of the commercially available engines and that is also applicable in the tool's clouds. It consists on sampling a mipmap [8] depending on distance. For the tool it required to manually generate mipmaps for each of the textures passed to the shader and manually set the LOD level based on the sample distance from the camera as engines only do this automatically for geometry.

Minor Shader Optimizations. The cloud rendering shader has been optimized removing certain elements and behaviours that decreased the performance:

- ***Dynamic Branching:*** That is, if or if-else statements with conditions that change at runtime and cause the shader to break parallelism by forcing the GPU to perform different calculations at the same time.
- ***Duplicated Operations:*** redundant operations that were calculated for every pixel and only needed to be calculated once have been moved out of the shader and passed as uniform variables instead, reducing the number of operations. Operations that were calculated more than once per pixel have also been moved to only be computed once and passed as variables on the methods.

[8] A collection of bitmap images that accompany a texture to increase rendering speed and reduce rendering artefacts. Each bitmap image in the set is a scaled-down version of the main texture.

These minor optimizations has required most parts of the shader code to be refactored to some degree but has increased performance in return.

Memory Related

Optimizations explained here focus on reducing VRAM usage on the GPU and making the size of variables in the shaders smaller so they take up less space and are easier and faster for the GPU to fetch.

Reduced Texture Bit Depth. The original RGBA textures stored 16 bits per channel, with each channel capable of holding one of 65,536 distinct values. The texture bit depth has been reduced to 8 bits, or 256 values, per channel with no significant visual change to the shape of the clouds. This makes the textures take less space and increases performance.

Blue Noise as Single Channel. Blue noise is a black and white texture and prior to the optimization phase it was encoded as a normal RGBA texture taking 4 channels worth of space. That has been changed and now all the blue noise values are encoded in a single red channel of a texture.

Base Noise Texture Size Reduced. The default size of the 3D base noise texture that forms the base shape of the clouds has been changed to be 128px * 128px * 128px instead of 256px * 256px * 256px with no noticeable visual changes but a performance increase. Users can still customize the size of this texture as desired.

Overview

Before any optimization, the cloud rendering shader performed at 93ms per frame on average, peaking at 131ms in certain cases on the target hardware. These values made it unusable for real time applications. After the optimization process, the average time it takes for a frame to render is 11ms with values as low as 4ms with some weather presets. This result represents a nearly tenfold increase in performance and makes it usable for videogames running at 60 fps (16ms). The table below details the performance increase with the different optimizations explained in this section applied:

© Oscar Pérez Martín 2022

Table 9. Optimizations performance

Optimization	Average ms (of all weather presets)
No optimization	93 ms
Reduced Texture Bit Depth + Blue Noise as Single Channel + Base Noise Texture Size Reduced	61 ms
Dynamic Steps with Density	35 ms
Dynamic Steps Over Distance + LODs Over Distance	19 ms
Minor Shader Optimizations	11 ms

© Oscar Pérez Martín 2022

Conclusions & Future Work

Finally, after months of development, a usable tool has been created, allowing users to author their own cloudscapes. All of the general objectives proposed in the Goals section have been accomplished:

Create a Tool. A free to download and use tool for Unity has been created, and its development has been documented in this thesis. Most of the tool's options can be accessed from Unity's inspector window, making it more accessible to people familiar with the engine but unfamiliar with code.

Develop a Project. The project has been completed, ending with a finished product that can be used by the general public; its planning, research and development are documented in this thesis.

Contribute Knowledge. Knowledge is being provided to those interested in developing a similar project or systems in two ways: through this thesis, with explanations of the tool and its systems and how they work, and through the tool itself, as the code is available both in GitHub and in the scripts contained in the tool.

The level of accomplishment of the specific objectives mentioned in the Goals section is discussed below:

Texture Generation. A system capable of generating seamless 3D textures has been developed to create the clouds. It has also been expanded to be able to make 2D weather maps procedurally. Users can tweak the generation of their 3D textures to customize the shape of their clouds.

Cloud Generation and Rendering. The Density and Lighting models are the most critical systems in the tool, as they define the look and feel of the entire cloudscape. A lot of time, effort and polish has gone into these systems to achieve the realistic look of the clouds in the tool. A high level of accomplishment has been reached with these systems; however, with the final light integration used to render the clouds it has not been possible to simulate the dark edge effect when looking away from the sun.

Customize the Clouds. The tool offers a high level of customization with two modes: the simple mode allows the user to choose from 4 premade weather presets and tweak them; the advanced mode has more options and lets the user customize everything about the look of the clouds.

Create and publish the tool. The tool development has been completed, it has been submitted to the Unity Asset Store as a package and it is currently in the process of being published.

Document Performance. The performance and efficiency of different methods and systems in the tool has been discussed and documented in the Development section of this thesis. To help document performance better, a benchmarking system has been developed with the tool, recording various performance statistics while executing the demonstration application and also working in Unity's play mode. Other tools to document performance have also been used to complement this system and are also mentioned in the Optimization subsection in the Development section.

Create a Demo. A small demonstration application has been created, with four versions of the application available in the Releases section of the project's GitHub repository. It allows the user to freely move around the world and customize the most important parameters of the tool in real time. It also shows its performance in real time and allows the user to further analyse it through its benchmark system.

In summary, all major objectives set have been accomplished, successfully developing a tool to create clouds with a very high level of customization. However, there have been some challenges during the development phase: some small features have been left out in favour of completing critical features with the high quality needed to accomplish the goals that had been set. This has been detailed in the Planning Changes and Deviation subsection in the Planning section. The tool also required a lot of optimization to be capable of performing the rendering in less than 16ms and be considered real time in the target hardware.

The development of the project shows that with current commercially available hardware it is possible for new generation games to use volumetric cloudscapes with a high level of realism. However, only a few games nowadays use this technique and with this tool the aim is to spread the knowledge and technology and give developers in the industry more options to work with.

The objective in the near future is to, once it gets approved and thus published to Unity's Asset Store, keep working on the tool, improving it and adding new features, as it can be very useful to Unity users. Developing a version of the tool for Unity's URP and

HDRP render pipelines would allow the project to reach more users and it is also being contemplated.

References

Bauer, F. (2019). Creating the Atmospheric World of Red Dead Redemption 2: A Complete and Integrated Solution. Retrieved from https://advances.realtimerendering.com/s2019/slides_public_release.pptx

Fong, J., Wrenninge, M., Kulla, C., & Habel, R. (2017). *Production Volume Rendering SIGGRAPH 2017 Course.* Retrieved from https://graphics.pixar.com/library/ProductionVolumeRendering/paper.pdf

Häggström, F. (2018). *Real-time rendering of volumetric clouds.* Retrieved from https://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf

Hillaire, S. (2016). *Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite.* Retrieved from https://media.contentapi.ea.com/content/dam/eacom/frostbite/files/s2016-pbs-frostbite-sky-clouds-new.pdf

Perlin, K. (2002). Improving Noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (pp. 681-682).

Schneider, A. (2015). The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn. Retrieved from http://advances.realtimerendering.com/s2015/The%20Real-time%20Volumetric%20Cloudscapes%20of%20Horizon%20-%20Zero%20Dawn%20-%20ARTR.pdf

Schneider, A. (2016). Real-Time Volumetric Cloudscapes. In W. Engel, *GPU Pro 7: Advanced Rendering Techniques* (pp. 97-127). CRC Press.

Schneider, A. (2017). Nubis: Authoring Real-Time Volumetric Cloudscapes with the Decima Engine. Retrieved from http://advances.realtimerendering.com/s2017/Nubis%20-%20Authoring%20Realtime%20Volumetric%20Cloudscapes%20with%20the%20Decima%20Engine%20-%20Final%20.pdf

Wrenninge, M., Kulla, C., & Lundqvist, V. (2013). Oz: The Great and Volumetric. *SIGGRAPH '13: ACM SIGGRAPH 2013 Talks.* Anaheim.