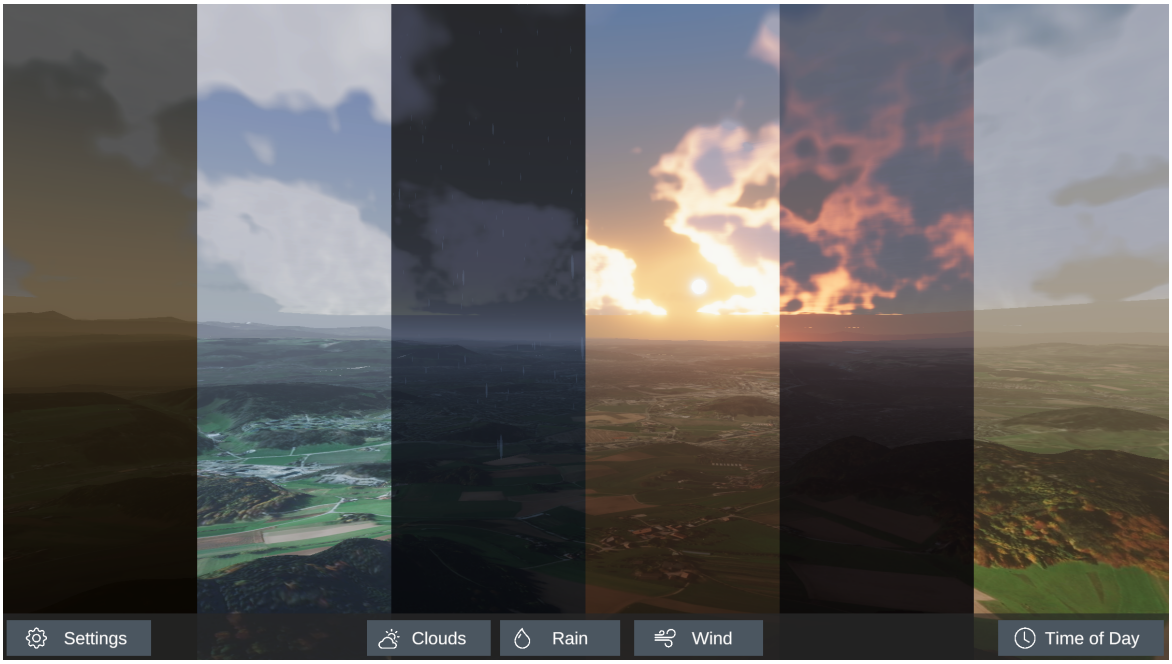
Near Real-time Weather Rendering System

Project documentation



Field of Studies:	BSc in Computer Science
Specialization:	Computer perception and virtual reality
Author:	Matthias Thomann
Supervisor:	Prof. Urs Künzler
Date:	June 10, 2021
Version:	1.0

Abstract

Clouds contribute a great deal to the overall ambience in games, but an implementation of such effects often proves to be more challenging than anticipated. To get as close as possible to real clouds, this project engages in researching and developing a near real-time weather rendering system. This means that real weather forecasts from *meteoblue* are used to visualize past, current and future weather at any given time of day. The environment is created with elevation model data from *ArcGIS*. Live photographs from *Roundshot* cameras can be viewed side-by-side with the rendered output.

The document dives into the science of clouds and illustrates the ten distinct classifications and how each of those could be represented in a weather simulation. In order to achieve high fidelity, the implementation relies on concepts like Voronoi noise generation and ray marching, which means to generate a random 3D cloud pattern and render it in volumetrically.

At last, the goal of the project is to create a fully fledged, near real-time weather rendering system in Unity. It is able to render procedural and volumetric cloudscapes, for any given date and time. A intuitive user interface allows the user to control the weather simulation manually or let it run automatically based on the *meteoblue* weather reports.

For future work, the weather rendering system could be incorporated in a game or further improved to achieve even higher visual realism.

Acknowledgements

Firstly, I want to thank my tutor Prof. Urs Künzler for his outstanding guidance and ongoing support during this project.

Next, I want to thank Dr. Eric Dubuis for supervising this work as my expert and for giving valuable feedback at the start and end of the project.

Further, I want to thank the team of *meteoblue* for the pleasant and unobstructed cooperation. I especially thank Mr. Edgar Caspar, with whom I communicated to most.

Contents

1 General

1.1 Purpose

During this project, all gathered information and knowledge about the researched algorithms and techniques are written down. All prototypes and the final results are documented and compared with real photographs of clouds.

1.2 Audience

This document is written with the intent to further expand existing knowledge about the topic, hence it requires a fundamental knowledge about computer graphics and rendering.

1.3 Revision History

Version	Date	Name	Comment
0.1	March 25, 2021	Matthias Thomann	Initial draft
0.2	April 05, 2021	Matthias Thomann	Cloud classification added
0.3	April 11, 2021	Matthias Thomann	Weather fronts added
0.4	April 12, 2021	Matthias Thomann	Implementation approach added
0.5	April 13, 2021	Matthias Thomann	Alternative approach added
0.6	April 30, 2021	Matthias Thomann	Noise generation added
0.7	May 02 , 2021	Matthias Thomann	Implementation approach reworked
0.8	May 08 , 2021	Matthias Thomann	Compute shader chapter reworked
0.9	May 09 , 2021	Matthias Thomann	Compute shader architecture added
0.10	May 10 , 2021	Matthias Thomann	Noise generation reworked
0.11	May 20 , 2021	Matthias Thomann	Technical implementation added
0.12	May 30 , 2021	Matthias Thomann	Project management added
0.13	May 31 , 2021	Matthias Thomann	Code snippets updated
0.14	June 03 , 2021	Matthias Thomann	Architecture & anatomy added
0.15	June 06 , 2021	Matthias Thomann	Results added
0.16	June 07 , 2021	Matthias Thomann	Testing chapter added
0.17	June 07 , 2021	Matthias Thomann	Polishing, Abstract reworked

2 Natural Clouds

Clouds are a substantial part of Earth's weather. They provide shade from the glistening sun on hot days and reflect the heat at night, keeping the ground warmer. Even for a layman, clouds are comprehensible and useful indicators for telling the weather. If they are dark and low-hanging, they usually bring rain. If they are puffy and scarce, they predict fair weather ahead.

2.1 Convection

In meteorology, convection describes the event of atmospheric motions in the vertical direction. Hot air rises from Earth's surface in form of bubbles, which are called *thermal columns* or just *thermals*. As the altitude increases, the thermal's air cool down. At some point, the warm air diluted by the surrounding colder air, after which its moisture condenses and starts forming clouds [1].

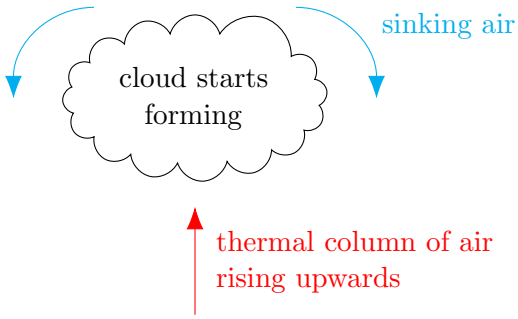


Figure 1: Lifting by convection.

Typically, thermal columns occur when sunlight is warming the ground, and thus the air directly above it. However, it can also be produced by the movement of weather fronts.

2.2 Weather Fronts

According to *metoffice* [2], weather fronts are boundaries between two air masses. Those masses differ in temperature, wind direction and humidity. There are three major types of weather fronts: *warm*, *cold* and *occluded* fronts.
In the following graphics, the warm front is marked with w_1, while the cold front is marked with c_1.

2.2.1 Precipitation Along a Warm Front

When a warm front approaches a cold front, it is likely that the impending clash results in clouds, bringing precipitation. The warm front carries warmer air and therefore rises over the colder, denser air. By advancing towards a cold front, the warm front pushes its warmer air higher, which means that thermals are created and clouds start to form [3].

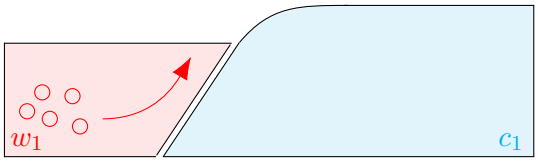


Figure 2: Warm front: warmer air advances, rising over the colder air, cooling down in the process.

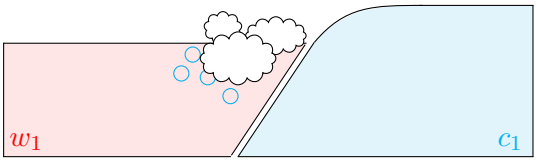


Figure 3: Warm front: as the air cools down, the moisture condenses. Clouds start to form.

2.2.2 Precipitation Along a Cold Front

A cold front represents the boundaries of an air mass carrying cold air. Like to the warm front movement, a cold front catching on a warm front can just as much produce clouds with precipitation. When trailing a warm front, thermals are produced in a similar way. As colder air is denser than warmer air, it pushes underneath it. By pushing up warm air, that air cools down as it rises, thus clouds start to develop [4].

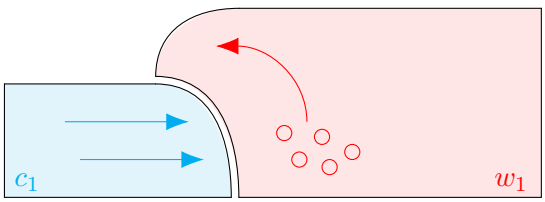


Figure 4: Cold front: colder air advances, pushing the warmer air upwards, cooling it down.

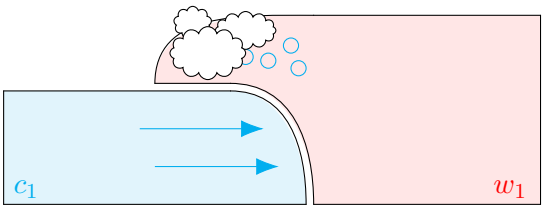


Figure 5: Cold front: as the air cools down, the moisture condenses. Clouds start to form.

2.2.3 Precipitation Along an Occluded Front

There is also the phenomenon of front occlusion, producing *occluded front*. This happens when there is a warm front that is caught in the middle of two faster moving cold fronts. At some point, the preceding cold front overtakes the warm front and forces it upwards, causing thermals of warm air rising. Depending on which of the two cold fronts is colder, the outcome may change. The milder cold front is denoted with c_1, while the other cold front with much cooler air is denoted with c_2.
If the preceding cold front carries cooler air than the succeeding, the occlusion is called a *cold occlusion* [5].

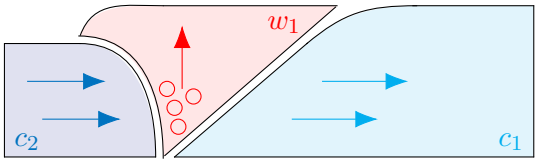


Figure 6: Cold occlusion: cool air catches up with a preceding cold front, forcing the warmer air in-between to go up, creating a thermal.

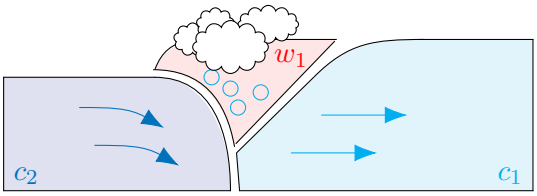


Figure 7: Cold occlusion: the cool air pushes underneath both other fronts. An occluded front is created, bringing heavy precipitation.

However, if the succeeding cold front is carrying cooler air than the preceding cold front, the occlusion is called *warm occlusion* [5].

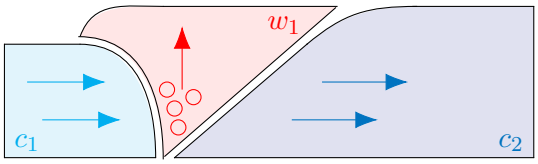


Figure 8: Warm occlusion: a cold front catches up with a warm front preceded by cool air, forcing the warmer air in-between to go up, creating a thermal.

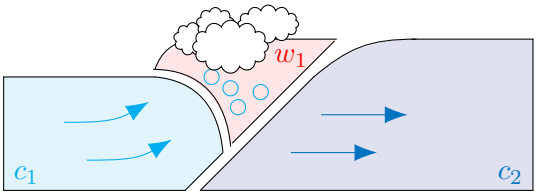


Figure 9: Warm occlusion the cold front is forced to climb over the cool air, pushing the warm front up. An occluded front is created, bringing heavy precipitation.

2.3 Classifications

In order to create a weather rendering system that is able to display many different cloudscapes, all types of clouds have to be understood first. The World Meteorological Organization (WMO) describes ten distinct cloud classifications. For each of those, there are further subtypes. For simplicity, those subtypes will be disregarded in this project.

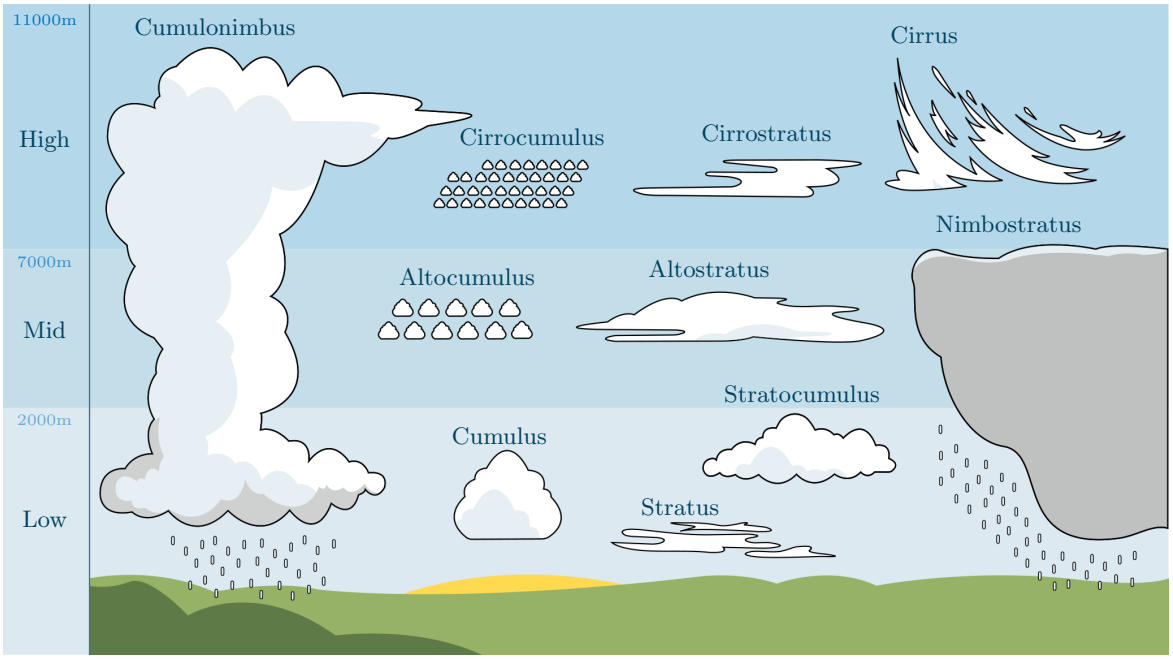


Figure 10: Distinct classifications of cloud shapes in the troposphere [6].

This graphic above provides and excellent overview of all distinct cloud types. Each type is depicted in its signature shape and labeled with its scientific name. Natural clouds are typically identified by two major factors: shape and altitude. The altitude, which is the distance from sea level to the cloud, is further split into three categories "low", "mid" and "high". This corresponds to the altitude at which the cloud usually forms, up to eleven kilometers above ground.
All of those clouds are formed in the troposphere, Earth's lowest atmospheric layer. Certain clouds may occur in the stratospheric or even the mesospheric layer, but they are usually a rare sight. Therefore, those clouds will not be covered in this project.

2.3.1 Cirrus

Cirrus clouds consist of thin, hair-like strands. They fall into the "high" altitude group and mostly appear in a bright white color, although they may take on the colors of the sunset or sunrise. Typically, they are formed when water vapor undergoes desublimation, the process in which gas turns into solid. This occurs when the water vapor freezes rapidly at high altitudes, turning into ice crystals.
However, cirrus clouds can also form from air that flows outwards of thunderstorms.

Figure 11: Cirrus clouds [7].

Interpretation: Fair weather, but they might announce the arrival of warm front in 12-24 hours, which is often preceded by rain several hours in advance. Even though cirrus clouds indicate precipitation, they themselves do not produce rainfall [8].

2.3.2 Cirrostratus

Cirrostratus clouds are similar to the cirrus clouds, only that they are even thinner. Those clouds depict more of a veil than a single cloud shape. They form under the same conditions as the cirrus clouds and can cover a massive area of the sky, spanning thousands of kilometers.
Cirrostratus clouds sometimes produce white rings or arcs of lights around the sun or the moon called the *halo phenomenon*. Sometimes, the cirrostratus clouds are so thin that the halo is the only way to tell if there are cirrostratus clouds.

Figure 12: Cirrostratus clouds [9].

Interpretation: Fair weather, but they indicate a warm front within one or two days, bringing precipitation [10].

2.3.3 Cirrocumulus

Similar to the other clouds of the cirrus family, the cirrocumulus are composed of ice crystals and formed at high altitudes. They are made up of many small, white, puffy clouds called *cloudlet*s. Their woolly look give the cloud the name suffix *cumulus*.
Cirrocumulus clouds are relatively rare, as they are naturally only formed when a turbulent vertical current meets a cirrus cloud layer. The cirrus cloud then disperses into many cloudlets.

Figure 13: Cirrocumulus clouds [11].

Interpretation: They do produce precipitation, but it never reaches the surface, meaning that cirrocumulus clouds are typically associated with fair weather [10].

2.3.4 Altostratus

The name for this grey, uniform sheet of clouds consists of the Latin words *alto* (elevated) and *stratus* (layered), summing up their appearance accurately. Altostratus clouds usually cover the whole sky and form a dull blanket of monocolored clouds with very few features. The sun- or moonlight may shine through them, but will most likely not be strong enough to cast hard shadows.

Figure 14: Altostratus clouds [10].

Interpretation: Altostratus clouds usually indicate precipitation, even more so if they are are preceded by cirrus clouds. If the precipitation increases in persistence and intensity, the altostratus clouds will lower and thicken into nimbostratus clouds.

2.3.5 Altocumulus

As with the cirrocumulus clouds, altocumulus clouds consist of small, puffy, white and grey cloudlets. These cloudlets are slightly bigger than the ones of the cirrocumulus cloud. It is easy to tell them apart, as the altocumulus cloudlets are usually more grey than white and are shaded on one side. Altocumulus clouds can form through the dispersion of altostratus clouds or through convection.

Figure 15: Altocumulus clouds [12].

Interpretation: Usually, they are found in settled weather. They do not produce precipitation that reaches the surface.

2.3.6 Nimbostratus

The nimbostratus clouds are the vast, grey clouds that bring heavy rain or snow for a longer period of time, sometimes up to multiple days. With their dark and gloomy appearance, they convey a dreary mood along with the persistent precipitation.
The thick, featureless layers of cloud are often formed by occluded fronts, when an altostratus starts lowering and gets denser [13].

Figure 16: Nimbostratus clouds [10].

Interpretation: They bring long-term rain or snow for several hours or days.

2.3.7 Stratus

Stratus clouds are low-layer clouds that usually only form in calm, stable conditions. They are often described as "high fog" as the have similarities in appearance.
Stratus cloud are formed by cool, moist air that is raised by mild wind breezes.

Figure 17: Stratus clouds [10].

Interpretation: They indicate quiet weather conditions, but sometimes produce sprinklings of rain.

2.3.8 Cumulus

Probably the most picturesque type of cloud is the cumulus. Its cotton-like look along with the soft, white color make it appear like candy in the sky.
The individual heaps of cumulus clouds remain strictly separated. The edge of each cloud is fuzzy and may change constantly.
Cumulus clouds are almost exclusively formed by convection. This is why they are a good indicator for gliders and pilots that there are upward winds [10].

Figure 18: Cumulus clouds [10].

Interpretation: They indicate fair weather, but can develop into cumulonimbus clouds, if weather conditions allow it.

2.3.9 Stratocumulus

These low-layer patches of cloud consist mainly of water droplets, absorbing a lot of light, giving them a saturated grey color.
They are the most common clouds on Earth and usually occur over oceans, but also when there is a change in weather or when a layer of stratus cloud breaks up. This means that stratocumulus clouds can also be present near cold, warm or occluded fronts.
Stratocumulus do not produce precipitation themselves, but are formed in many different conditions, including rainy or calm weather.

Figure 19: Stratocumulus clouds [10].

Interpretation: They announce an instability of the atmosphere and are usually present before an occlusion of weather fronts.

2.3.10 Cumulonimbus

Cumulonimbus clouds are massive, high-towering heaps of cloud, spanning over the whole troposphere. Their top is often shaped like an anvil, whereas the base if flat and dark, giving them a menacing look. They are referred to as thunderclouds, because they are the only type of cloud that is able to produce hail, thunder and lighting [15].
Cumulonimbus clouds are formed trough natural convection or as a result of forced convection when a cold front pushes up warm air.

Figure 20: Cumulonimbus clouds [14].

Interpretation: They cause extreme weather like heavy torrential rain, hail storms, lighting and even tornados.

3 Implementation Approach

Clouds are comprehensible indicators for telling the weather. They offer many visible features to make an rough prediction of the weather conditions, or weather changes to come. As described in subsection 2.3, some cloud types only form under specific conditions. Also, whenever certain clouds are present, precipitation is shortly followed, as it is with altostratus clouds.
Those factors allow a prediction of the weather, but for this project, the process is reversed. The given data is not an image of clouds, but meteorological measurement data, and the desired outcome is not a prediction, but an image of clouds.

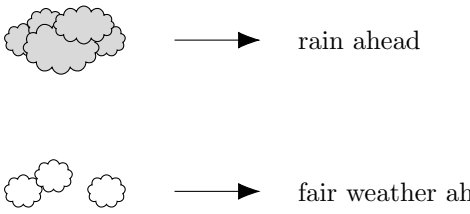


Figure 21: Weather information based on visual data.

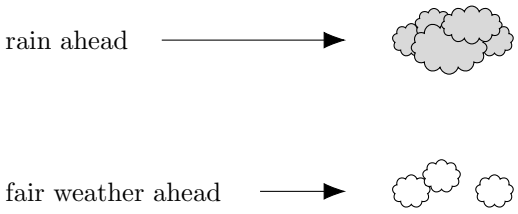


Figure 22: Visual construction based on weather information.

For any given day to render, an implementation would require data from that day but also from the near future of that day. So, in order to render a cloud image for day x, a potential algorithm could look like this. Note that the listing below describes only an idea and is by no means final or compulsory.

```
// weather data including 7-day forecast
WeatherData data;
CloudRenderer renderer;

function renderClouds(Day x) {
    if (x > TODAY + 7) throw;

    d1 = data.getDataFor(x);
    d2 = data.getDataFor(x + 1);
    d3 = data.getDataFor(x + 2);
    // and so on...

    // sophisticated checks about current and future conditions:
    if (d1.fairWeather && d2.fairWeather)
        return renderer.clearSky();
    if (d1.fairWeather && d2.isRaining)
        return renderer.cloudsOnclearDayBeforeRain();
    if (d1.isRaining)
        return renderer.cloudsOnRainyDay();
    if (d2.isRaining)
        return renderer.cloudsBeforeRainyDay();
    if (d3.isRaining)
        return renderer.clouds2DaysBeforeRainyDay();
    // and so on...

}
```

Listing 1: Pseudo-code of cloud render algorithm.

3.1 Look-Ahead Issue

The approach as described above relies on having data from a couple of days ahead of time. Assumed that number of days is t, then the weather data for day x could only be rendered t days after x. That would mean, for such an approach to work, the weather of today can not be rendered before t days later.
In this case however, the weather measurement data retrieved from *meteoblue* also contains a seven-day weather forecast. Given that t is less than or equal to seven and an implementation still produces accurate cloud imagery, it would no longer be an issue.

3.2 Layers of Cloud Shaders

As identified in subsection 2.3, most of the clouds in the troposphere only appear at certain altitudes. The high-level clouds are all of type *cirrus*. The mid-level types are altostratus and altocumulus, while the low-level types are cumulus, stratocumulus and stratus.
This leads to the conclusion that some of the cloud types could be consolidated into a combined layer and rendered by the same shader. Exception to that are only the two larger types that span over multiple height levels: nimbostratus and cumulonimbus. For those, a more unique solution has to be found.

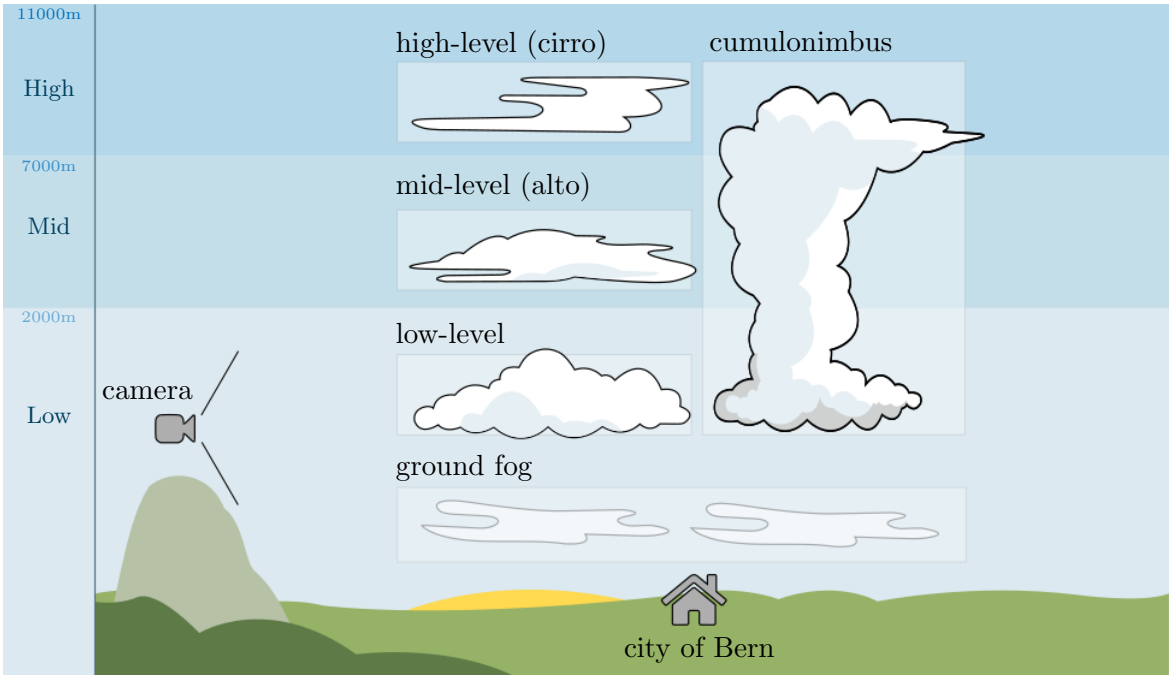


Figure 23: Layers of cloud shaders.

3.2.1 High-Level Clouds

The uppermost layer would contain cirrus, cirrostratus and cirrocumulus clouds. All of these form under similar weather conditions and closely resemble each other in appearance and formation.
A potential shader for that layer could be programmed to render a base variant of all three cirrus clouds, which would then be parametrized into each individual type of cirrus cloud, whichever is currently visible.

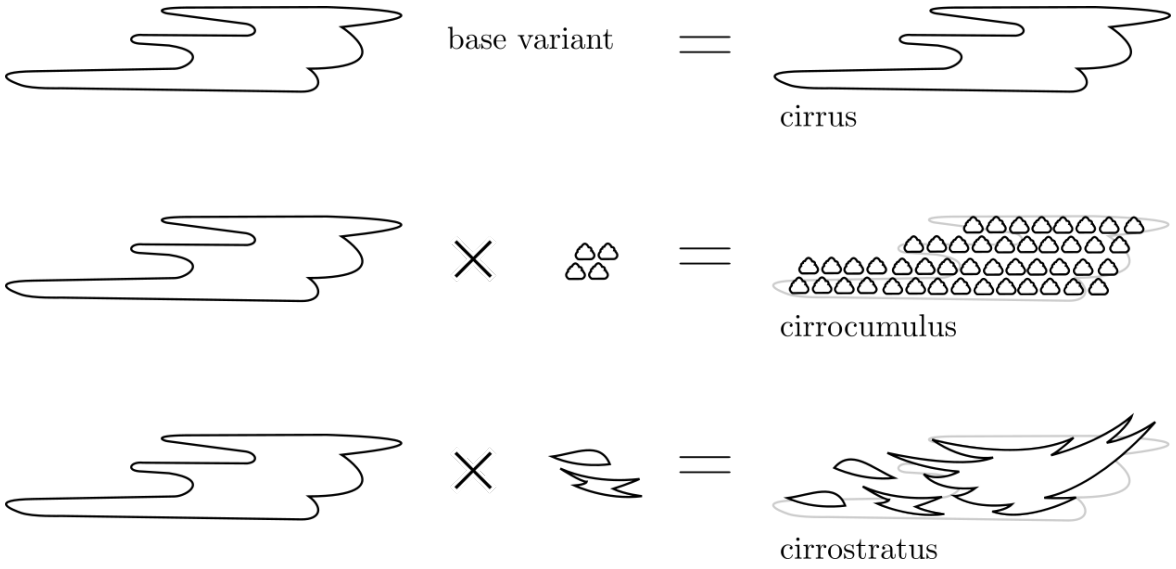


Figure 24: Breakdown of the highest shader layer.

3.2.2 Mid-Level Clouds

The middle layer consists of altostratus and altocumulus clouds, the latter mainly occurring due to dissipation of the former one. Since they have many shared characteristics, apart from the puffiness, they are predestined to be processed together.
Given a shader is flexible enough to render altostratus clouds, then it is most likely also able to render altocumulus, with only few adjustments necessary.

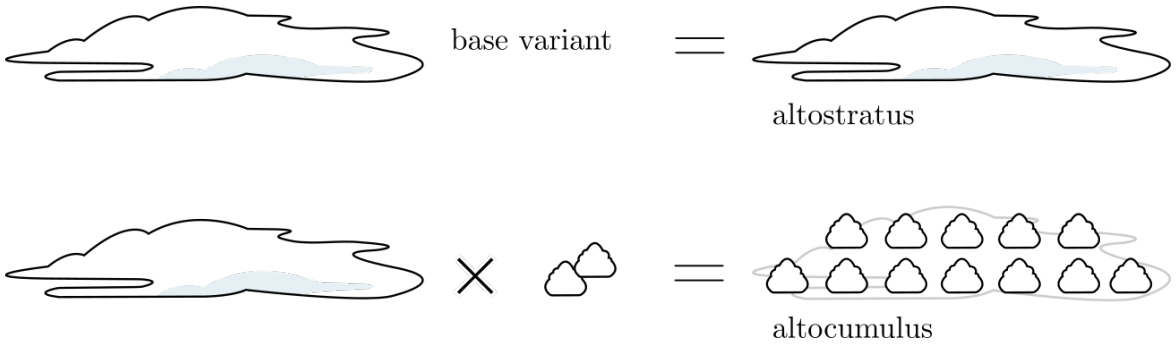


Figure 25: Breakdown of the middle shader layer.

3.2.3 Low-Level Clouds

The lower layer may prove to be more complex than the others, as the stratus and cumulus clouds do essentially not look alike. However, cumulus clouds could be described as less denser, smaller and separated instances of stratocumulus clouds.
If a shader would be able to render stratocumulus clouds and allow to control the density or the spreading of such, then cumulus clouds could be rendered in the same manner.

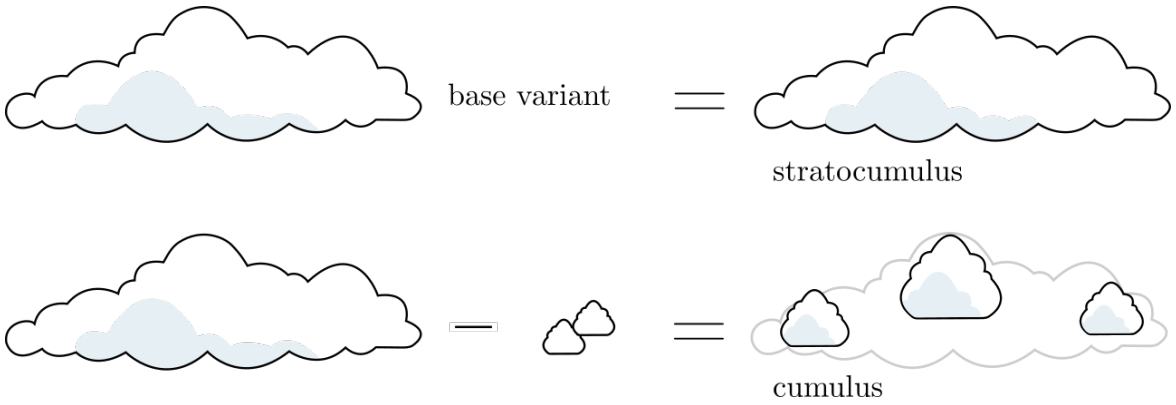


Figure 26: Breakdown of the lowest shader layer.

3.2.4 Ground Level Fog

The lowest layer consists of fog. It is conceivable that stratus clouds will also be placed in that layer. Both type of clouds are to some extent combinable, as they primarily vary in density.
Therefore, a shader would need to have control over the outcome's density and lightness for it to be able to render both stratus clouds and ground fog.

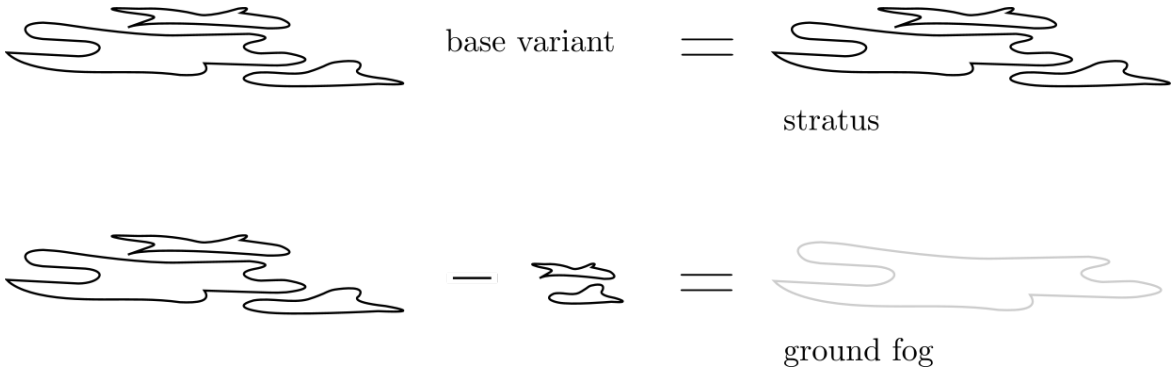


Figure 27: Breakdown of the fog shader layer.

3.2.5 Cumulonimbus Layer

With all the previous layers implemented, only the two large cloud types are left, one of them is the cumulonimbus.
Assuming the observer is walking directly underneath a cumulonimbus cloud, the cloud itself and its defining visual features are not really recognizable. It could easily be mistaken for other clouds that produce precipitation.

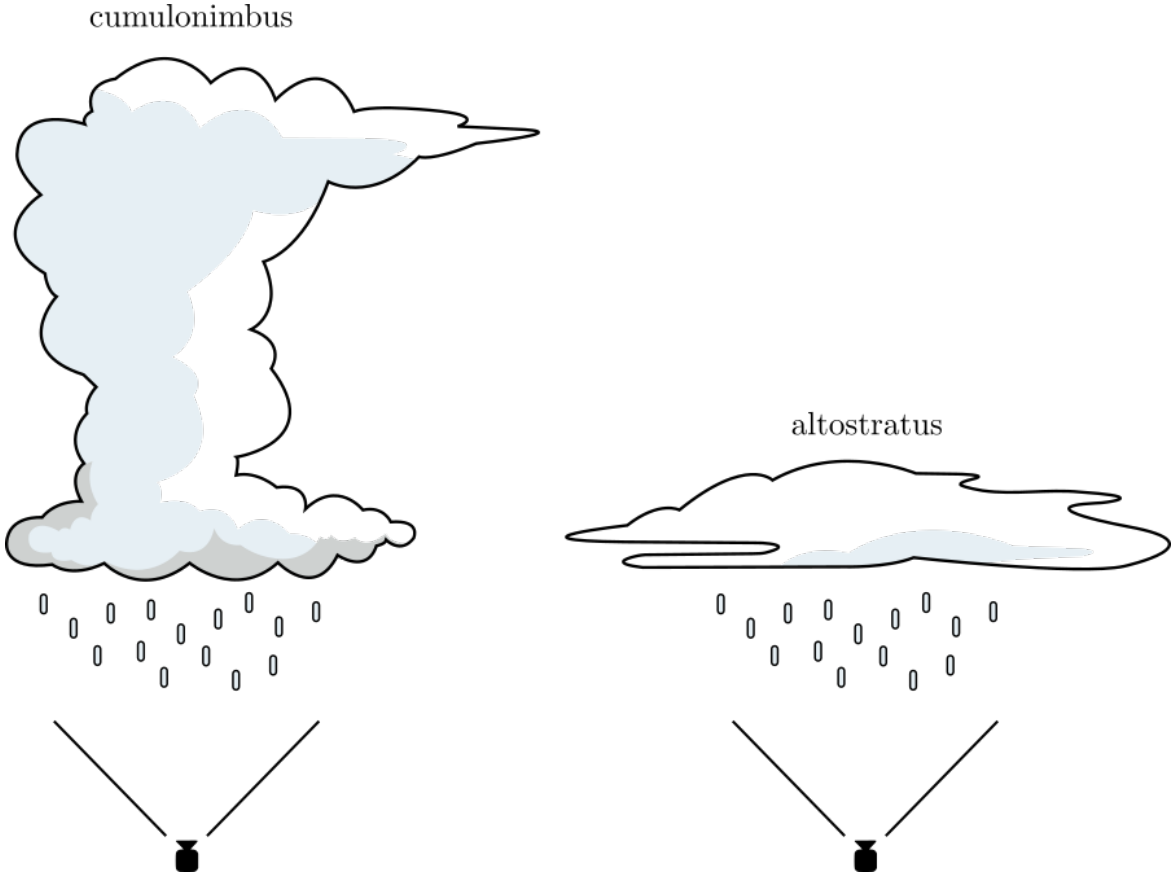


Figure 28: Perspective similarities under clouds with precipitation.

Under that assumption, it is considered that the cumulonimbus cloud will only ever be seen from a distance. This is why these clouds will most likely be rendered in their own layer, farther away from the main camera, but spanning over all other height levels.

3.2.6 Nimbostratus Substitute

A similar issue as with the cumulonimbus clouds is present for the nimbostratus clouds. Due to them being thick, dark layers of cloud lacking features and contours, they are more difficult to render.
It is, however, imaginable to omit the type nimbostratus altogether and substitute it by combining and tuning the already existing layers. For example, the nimbostratus cloud might be imitated by darkening the color of altostratus clouds as well making them thicker. With additional stratus clouds and increased fog density, a layman could probably no longer tell the difference.

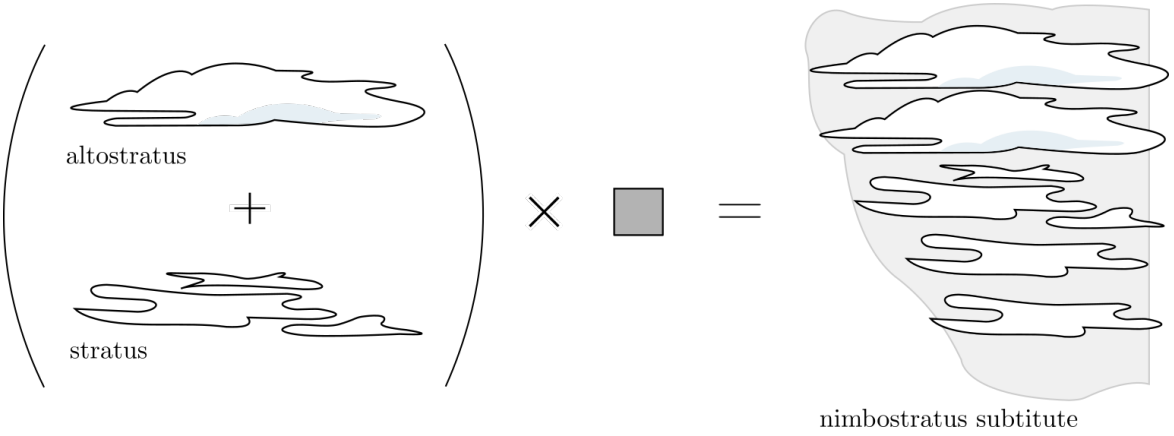


Figure 29: Breakdown of the nimbostratus substitute.

3.3 Exclusiveness Issue

All shader layers described in subsection 3.2, except the cumulonimbus layer, come with an issue. Given that multiple cloud types are consolidated into one layer, and that layer can only render one cloud type at a time, then no two cloud types of the same layer can ever be rendered together.
However, the chance for multiple clouds of the same layer to coexist is considered to be negligible, which is why the issue is disregarded in this project.

3.4 Background Weather Consideration

Clouds can be seen from great distances, especially when the observer is located on elevated terrain, which is the case in this project. For this reason, clouds in close proximity to the observer do not have to be alike more distant ones. This is also true for weather conditions. Because of this, whenever rendering cloudscapes from an angle where the horizon is also visible, the weather measurements from places in the far distance also have to be taken into account.
As mentioned in the bachelor project specification document, there are two points of view from which the weather will be rendered. They are two mountains near the city of Bern: the *Bantiger* and the *Gurten* mountain.
From both peaks, Bern lies in the center of the view. However, a city in the background of each perspective has been chosen to account for weather in the distance.

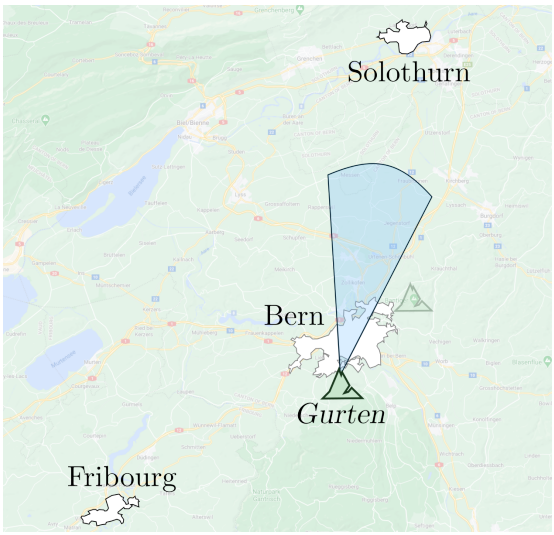


Figure 30: Perspective view from the Gurten mountain: distant weather in Solothurn is considered.

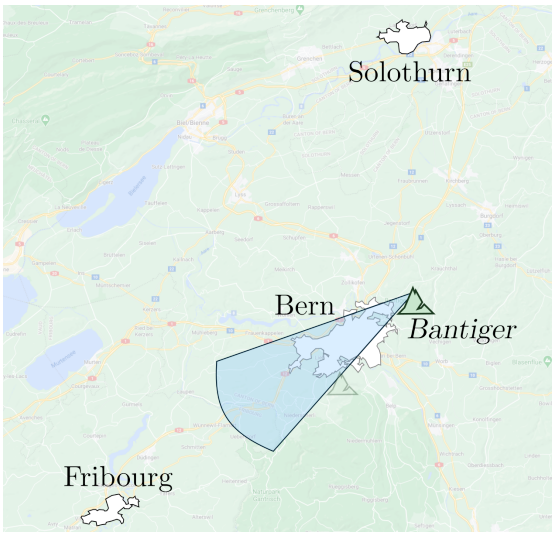


Figure 31: Perspective view from the Bantiger mountain: distant weather in Fribourg is considered.

3.5 Weather Data Interpolation

Following up on the background weather consideration, all shader layers also need to know those distances and adjust accordingly. Given there are two measurement sets, one for each city, then the weather in-between can be a combination of both sets. A very common method to achieve such an evaluation of interim data is called *linear interpolation*. Linear interpolation can be defined as a function *lerp*, if $0 >= t >= 1$:

$$lerp(a, b, t) = a + (b - a) * t$$

Assuming that the two weather measurement sets w_1 and w_2 can be interpolated, then the function *lerp* can be used. In the following example, w_1 represents fair weather in Bern, whereas w_2 represents cloudy and gloomy weather in Solothurn. With increasing distance along the axis L, factor t gradually rises from 0 to 1.

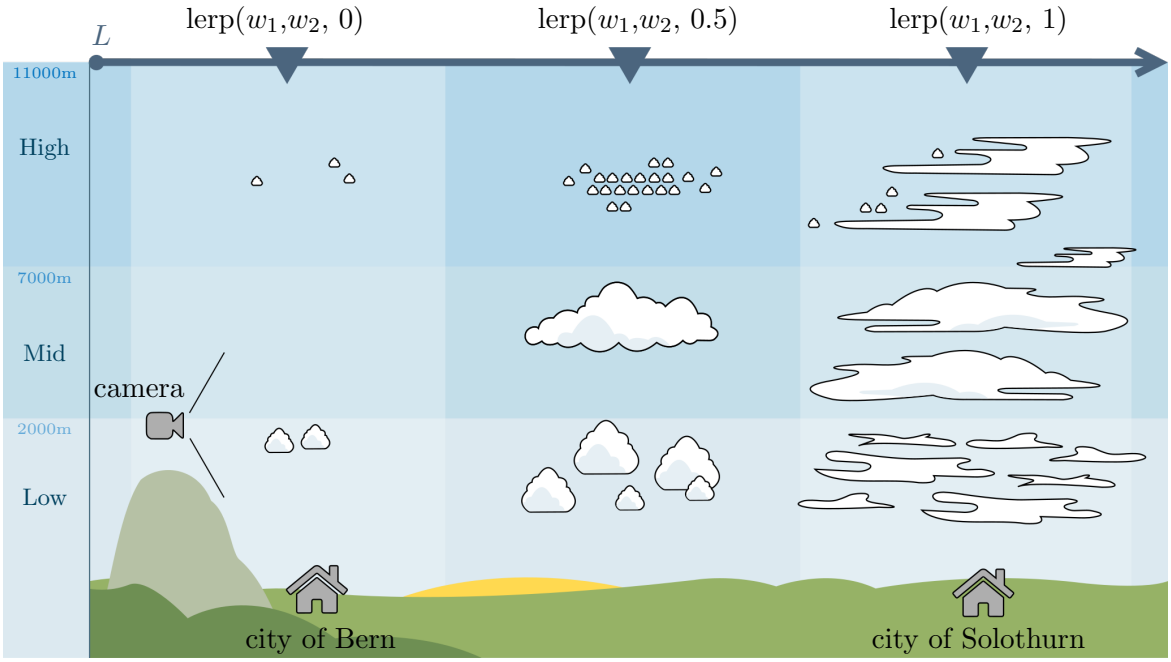


Figure 32: Weather data interpolation for the cloud layers from Bern to Solothurn.

In summary, at Bern ($t = 0$), weather for data set w_1 is displayed, while at Solothurn ($t = 1$), weather for the data set w_2 is rendered. Everything in-between is a mixture of both data sets, relative to which ever is closer.

3.6 Alternative Approach

Every subsection that describes another feature of the implementation approach increases the complexity of the developing concept, which is already depending on many ideas and structures to work. In cases like this it is best to prepare an alternative approach to implementation, that can be pursued should the primary one fail.

The core concept of this second implementation approach is based on particle systems. It still relies on having four major layers with an additional cumulonimbus block. The nimbostratus will also be substituted. Those layers are almost identical to the layers described in subsubsection 3.2.1 to subsubsection 3.2.5.
However, instead of the layers each being a single shader, they are replaced with a particle system that emits cubes of clouds. Those cubes are rendered by different shaders, depending on which layer they are spawned in. The highest layer, P_1, would emit cloud cubes of the cirrus family. The middle layer, P_2, is responsible to render alto cloud cubes. Layer P_3 would spawn low-level cloud cubes, while the lowest layer, P_4, would create fog particles.

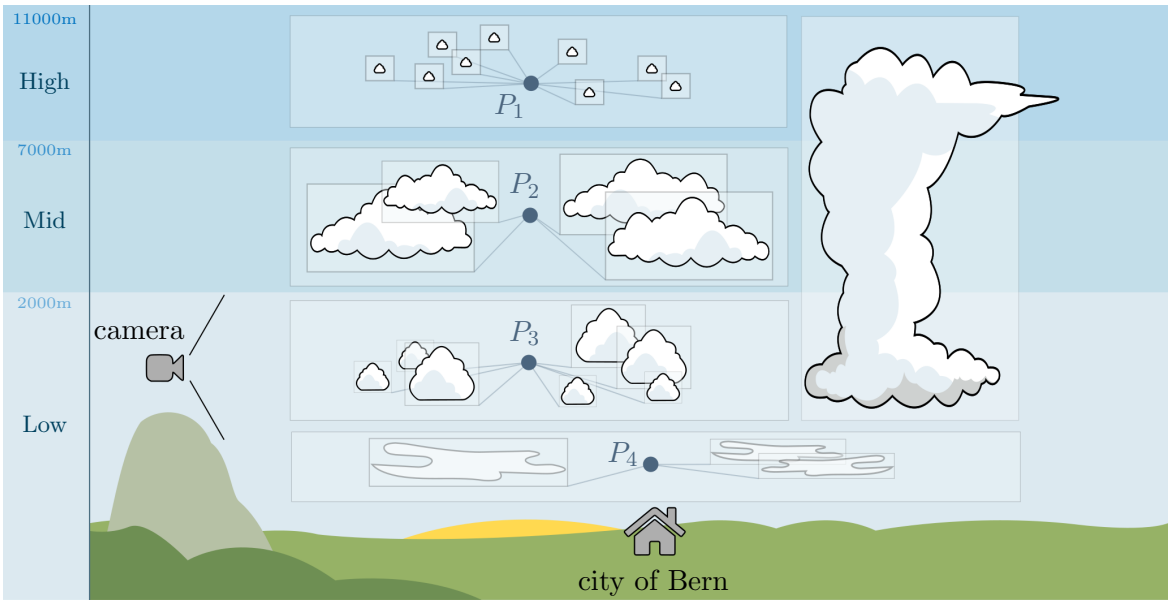


Figure 33: Alternative implementation approach based on particle systems.

This alternative approach would offer a much higher flexibility in controlling exactly how many clouds are present. Each layer is in absolute control of when it spawns cloud particles, how fast the spawn rate is, the particle's size and many more characteristics.
Nonetheless, contrary to the first approach, a linear interpolation of the clouds will prove to be more difficult in this case. A particle system cannot be interpolated along two positions, as it is bound to a fixed location. This issue be could be disregarded at the cost of realism, by only using three particle systems: one for Bern, one in-between both cities, using interpolated weather data, and one for the distant city.
Additionally, the number of cloud cubes may heavily impact performance. If the system is required to limit the number of cloud cubes, its realism would suffer further.

4 Noise Generation

Nature's chaotic and fortuitous behavior creates a world full of diversity and unpredictability. This can be observed in a surprisingly high amount of objects, structures and phenomena. For example, the following images show photographs of patterns that seem almost completely random.

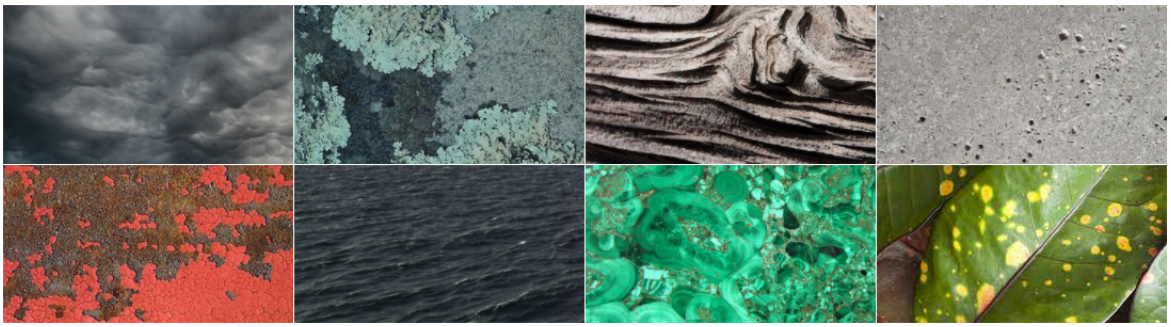


Figure 34: Random patterns observed in Nature [16].

In computer science, the virtual recreation of such randomness has been studied continuously over the last decades. The outcome of a randomness generator is called *noise*. There are many established algorithms to create random patterns, one of which generates the famous *Voronoi* noise.

4.1 Previous Work

Many of the following subsections rely on concepts and algorithms that have been thoroughly explained in the project's previous work [17]. This include sine-based deterministic number generation algorithms, also known as also *pseudo-random* number generation [18]. It further includes different noise generation algorithms like Perlin noise [19] and functions like the fractal Brownian motion [20].

Those algorithms will not be described again, as they have already been studied and documented before.

4.2 Voronoi Noise Algorithm

One of the more commonly used procedural pattern generation algorithms is that of Steven Worley, developed in 1996 [21], called *Worley*'s algorithm. The algorithm is also known as the *Voronoi* algorithm due to its similar appearance to a Voronoi diagram. In that diagram, points, called *seeds*, are randomly scattered inside a defined space. After that, regions are created, consisting of all points closer to that seed than to any other.
The Voronoi noise algorithm creates a cellular pattern and is therefore well suited for simulating natural distribution of cloud heaps, as they are in some way also arranged in cells.

The noise algorithm starts by dividing the space into a grid, for which each cell is assigned a random point. From there, each pixel gets shaded by how far it is to the seed in its cell.

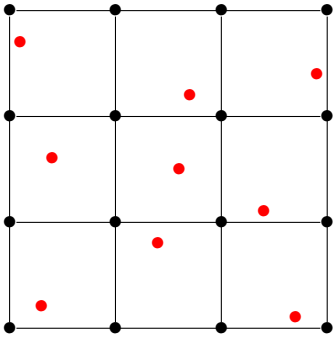


Figure 35: Voronoi grid with pseudo-randomly assigned seed points for each cell.

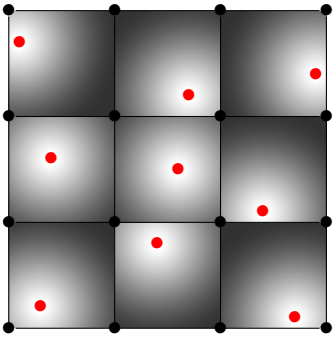


Figure 36: Voronoi grid with seed distances visualized.

As recognizable in Figure 36, hard contours are still visible along the grid lines. This can be improved by including the adjacent cells when finding the closest seed for any given fragment. This amounts to $3^n - 1$ neighboring cells, where n is the number of dimensions. This means for 2D space its eight cells, while in 3D its 26.

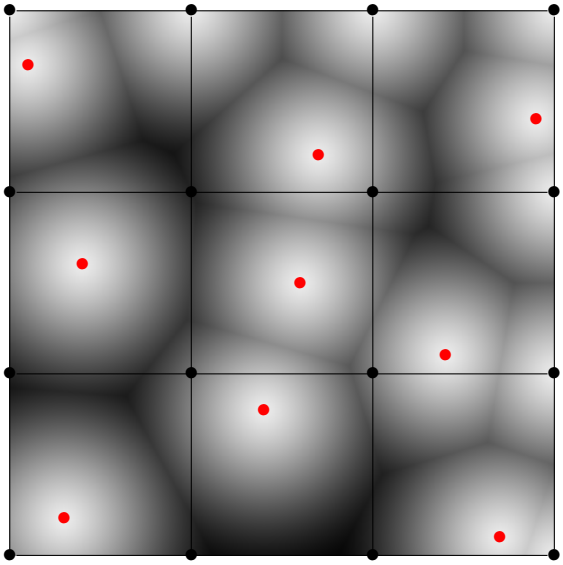


Figure 37: Complete 2D Voronoi noise texture.

An implementation in high-level shading language (HLSL) of this relatively simple algorithm could look like the following listing.

```
float2 randomSeed(float2 co) {
    return float2(
        fract(sin(dot(co, float2(12.9898, 78.233))) * 43758.5453123),
        fract(sin(dot(co, float2(39.3461, 11.135))) * 14375.8545359));
}

float voronoi(float2 p) {
    float2 baseCell = floor(p);
    float dMin = 999;

    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            float2 cell = baseCell + float2(x, y);
            float2 seed = cell + randomSeed(cell);
            float d = distance(seed, p);
            if (d < dMin) {
                dMin = d;
            }
        }
    }
    return dMin;
}
```

Listing 2: Implementation of 2D Voronoi noise algorithm.

The 3D equivalent of the algorithm looks fairly similar.

```
float3 randomSeed(float3 co) {
    return float3(
    fract(sin(dot(co, float3(12.989, 78.233, 37.719))) * 43758.5453123),
    fract(sin(dot(co, float3(39.346, 11.135, 83.155))) * 14375.8545346),
    fract(sin(dot(co, float3(73.156, 52.235, 09.151))) * 31396.2234116));
}

float voronoi(float3 p) {
    float3 baseCell = floor(p);
    float dMin = 999;

    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            for(int z = -1; z <= 1; z++) {
                float3 cell = baseCell + float3(x, y, z);
                float3 seed = cell + randomSeed(cell);
                float d = distance(seed, p);
                if (d < dMin) {
                    dMin = d;
                }
            }
        }
    }
    return dMin;
}
```

Listing 3: Implementation of 3D Voronoi noise algorithm.

4.3 Seamless Noise

In Figure 37, the generated cells appear to be randomly distributed, but there is still a major flaw in the texture.
As already mentioned previously, the noise generation is confined to a fixed region. This poses an issue when the desired area is larger than the defined space of the algorithm. It is therefore not possible to tile the same texture without visible seams in-between the tiles. Tiling the texture saves the calculation of an overly large area of noise and is therefore the desired approach in many cases.

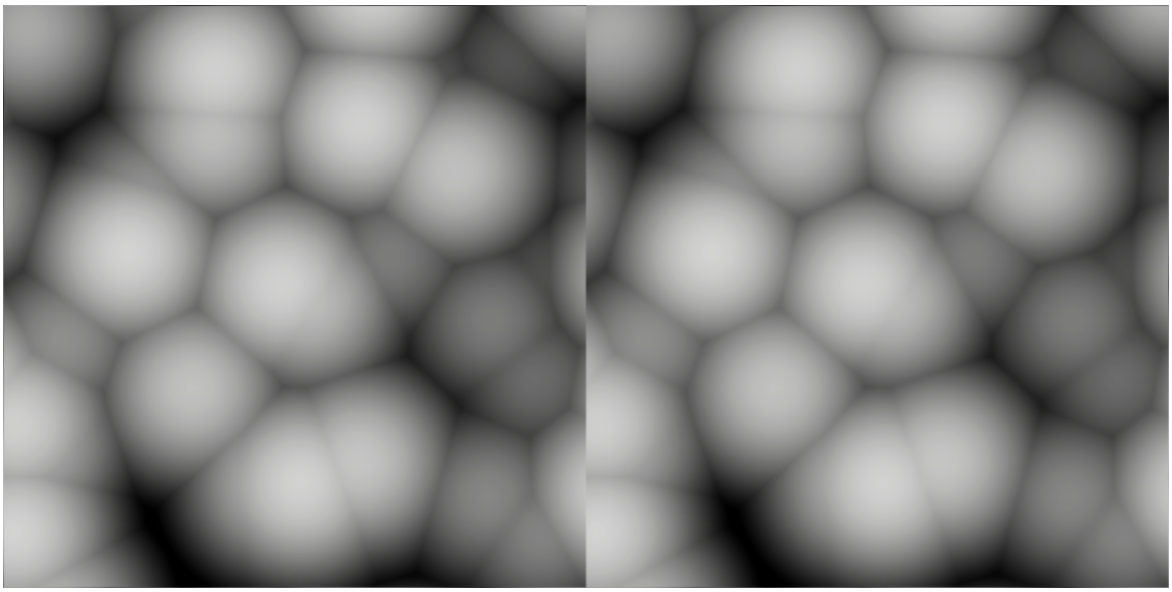


Figure 38: Tiled 3D noise texture slices.

The problem occurs when checking the neighboring cells of the current cell. For example, here are the seeds for two adjacent tiles of the same noise texture.

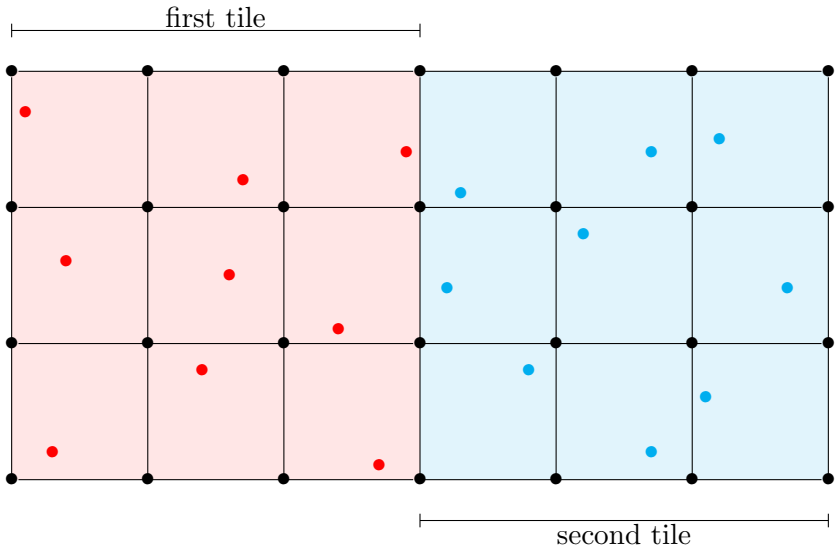


Figure 39: Voronoi noise seeds of the second tile are different than the seeds of the first tile.

When inspecting the cell $c_{2,1}$ of the first tile, some of its neighbors, namely $c_{3,0}$ to $c_{3,2}$, are part of the next tile. However, since the desired outcome is a repeating texture, the underlaying texture will be the same as the first one.

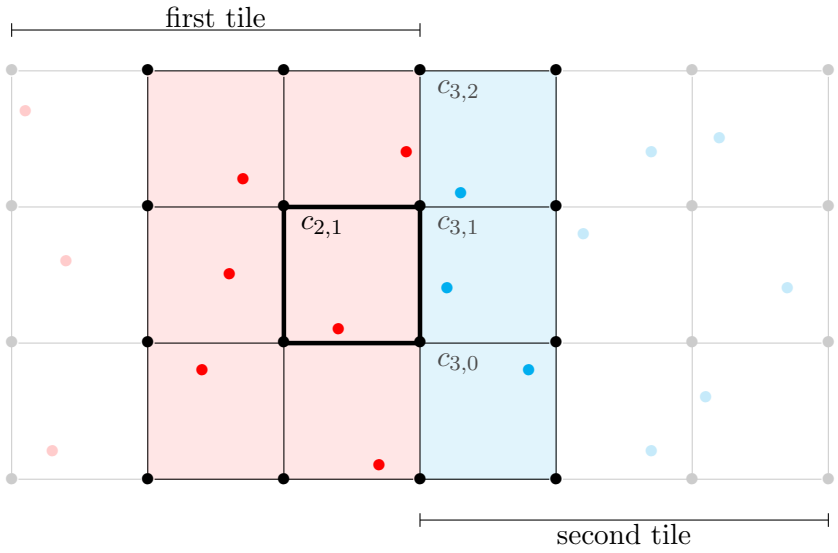


Figure 40: Voronoi noise seeds are sampled from the second tile.

Naturally, when using seeds from the second tile but placing the texture from the first tile underneath, a disruption in the pattern will be visible in the form of a seam. This is why, when the sampling point exceeds the defined space of the texture, the seeds will always be calculated based on the first tile again. Instead of sampling $c_{3,0}$ to $c_{3,2}$ from the second tile, $c_{0,0}$ to $c_{0,2}$ will be used.

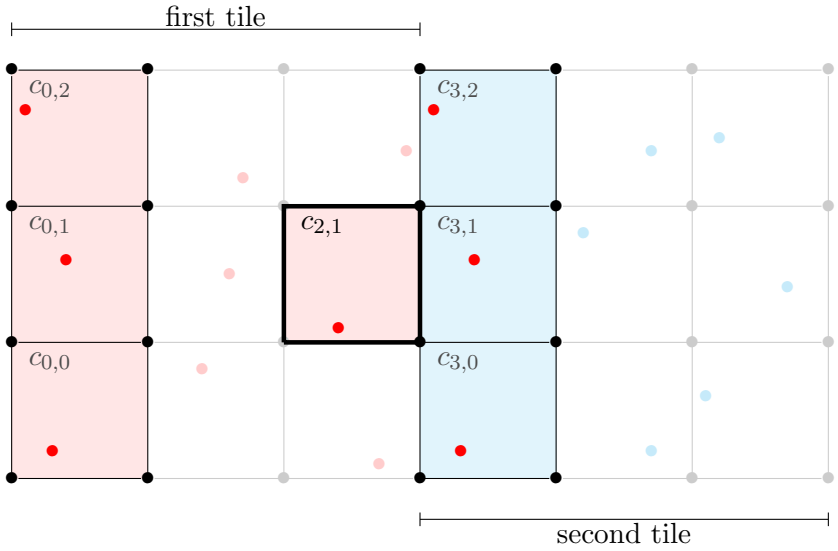


Figure 41: Voronoi noise seeds that exceed the boundary are sampled from the first set again.

In conclusion, the same set of points that is calculated for the first texture tile needs to be used again for every succeeding tile.

Mathematically, this is solved by using the *modulo* operation. The dividend is the current cell and the divisor is the period, or how often the noise texture is repeated.

```
float voronoi(float3 p, float3 period) {
    float3 baseCell = floor(p);
    float dMin = 999;

    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            for(int z = -1; z <= 1; z++) {
                float3 cell = baseCell + float3(x, y, z);
                float3 tiledCell = cell % period;
                float3 seed = cell + randomSeed(tiledCell);
                float d = distance(seed, p);
                if (d < dMin) {
                    dMin = d;
                }
            }
        }
    }
    return dMin;
}
```

Listing 4: Implementation of a partially seamless 3D Voronoi noise algorithm.

Interestingly, this does only partially solve the problem. As it turns out, the modulo operation available in HLSL treats negative values differently than expected. Instead of returning the absolute modulo value, it returns the remainder, including negative values.

$$\begin{aligned} \textit{expected} &: -5 \bmod 3 = 1 \\ \textit{actual} &: -5 \bmod 3 = -2 \end{aligned}$$

When used this way, the outcome still has visible seams where the modulo operations have to deal with negative numbers. The issue is further explained by Ronja [22].

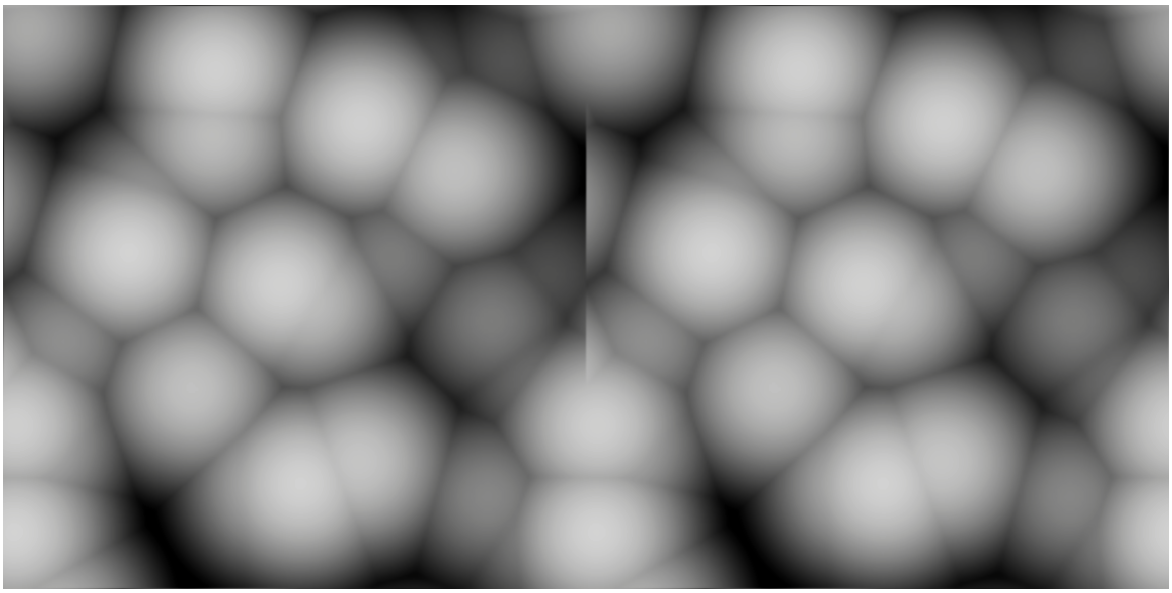


Figure 42: Partially seamless, tiled 3D noise texture slices.

The problem is easily fixed, though. To get a positive dividend, one can simply take the remainder of the dividend, add the divisor and then take the remainder again, knowing that the number will now be positive beforehand.

$$mod(x, y) = (x \,\%\, y + y) \,\%\, y$$

Alternatively, the implementation as described in the OpenGL documentation [23] can be used. It is capable of dealing with negative values, and is defined as follows:

$$mod(x, y) = x - y * floor(x/y)$$

```
#define mod(x,y) (x-y*floor(x/y))

float voronoi(float3 p, float3 period) {
    float3 baseCell = floor(p);
    float dMin = 999;

    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            for(int z = -1; z <= 1; z++) {
                float3 cell = baseCell + float3(x, y, z);
                float3 tiledCell = mod(cell, period);
                float3 seed = cell + randomSeed(tiledCell);
                float d = distance(seed, p);
                if (d < dMin) {
                    dMin = d;
                }
            }
        }
    }
    return dMin;
}
```

Listing 5: Implementation of seamless 3D Voronoi noise algorithm.

And with that, the noise texture tiles seamlessly.

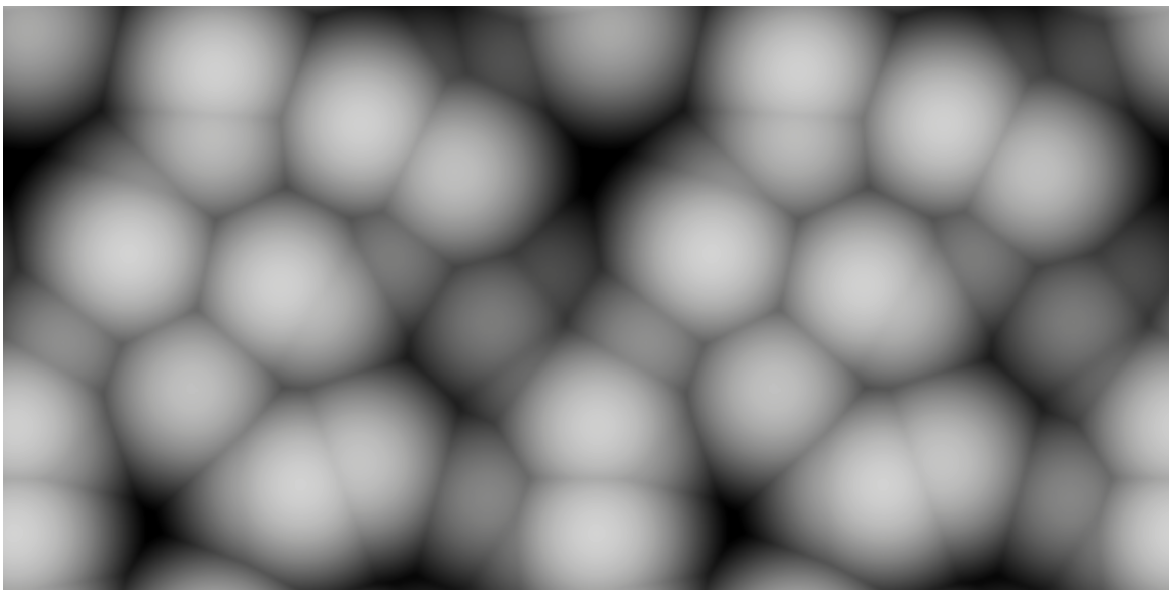


Figure 43: Seamlessly tiled 3D noise texture slices.

4.4 Compute Shaders

Compute shaders are special programs that run on a graphics card. They are used for high-speed general-purpose computing. Unlike regular shaders, they do not write directly to the frame buffer and require no setup of vertices. It is more common that the target buffer is a 2D or 3D texture.
Compute shaders also skip the render output pipeline (ROP), which is responsible for a process called rasterization, in which the final image is created. This process is usually computationally demanding and by skipping it, a great deal of performance can be saved.

4.4.1 Compute Shader Structure

The following figure shows that the compute shader is dispatched with a large number of thread groups, each containing a set of threads, of which each represents a single invocation of the shader.

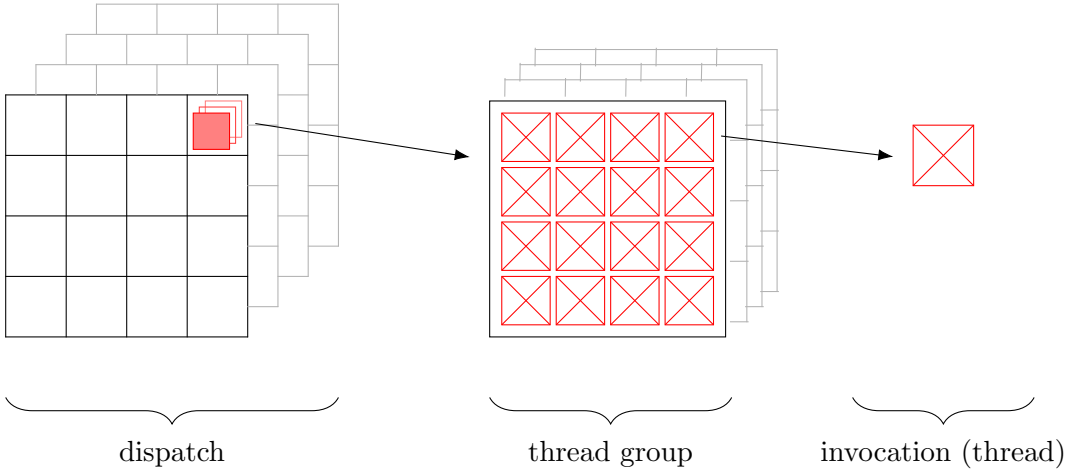


Figure 44: Hierarchical thread structure of compute shaders.

In Unity, compute shaders are written in HLSL and interact with Microsoft's DirectCompute technology, a graphics card interface for compute shaders. This is a standard example of such a shader:

```
1 #pragma kernel CSMain
2
3 RWTexture3D<float4> _Result;
4
5 [numthreads(4,4,4)]
6 void CSMain (uint3 id : SV_DispatchThreadID)
7 {
8     _Result[id.xyz] = float4(1,0,0,1);
9 }
```

Listing 6: A standard compute shader template.

The so-called *kernel* is defined on the first line and describes the method that will be executed when the compute shader is dispatched. In this example, there is only one kernel with the name `CSMain`. When executed, it writes to a 3D texture and sets the color of a specific 3D texture element, short *texel*, to red.
On the third line, a 3D texture variable is declared. It does not have to be initialized.

The type is `RWTexture3D`, which means the texture is enabled for both reading and writing. Another important part is the `[numthreads(4,4,4)]` statement. It specifies the dimensions of thread groups.

4.4.2 Thread Groups

When running a compute shader, the kernel is executed on many threads in parallel. This allows the GPU to split up the computation instructions declared in the shader. For each part, a thread group is created, which only handles that specific part. These thread groups run individually and simultaneously, speeding up the process massively.
The dimensions of a thread group can be controlled with the `numthreads` attribute. In the example above, for each thread group, there will be 4 threads in the X dimension, 4 threads in the Y dimension and 4 threads in the Z dimension.

Every kernel has a parameter called `id`. This is the identifier of the current thread, short *thread ID* or *dispatch thread ID*. The thread ID is unique over all thread groups. The following example shows how a thread group is structured.

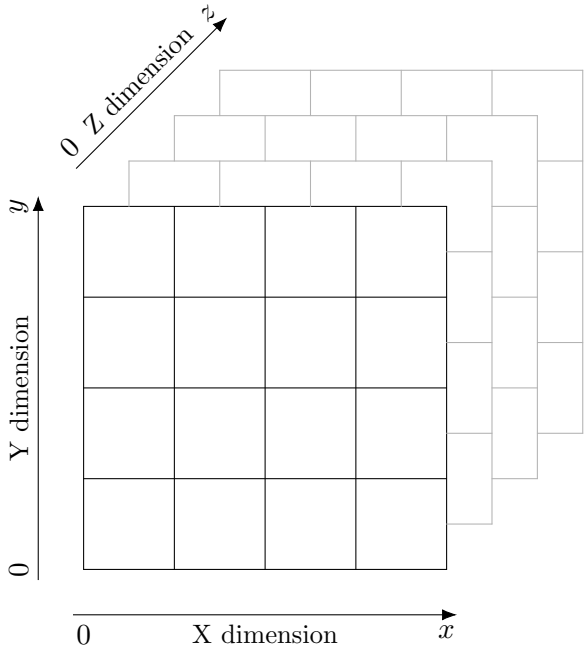


Figure 45: Compute shader thread group with labeled dimensions.

Each cell represents a single thread that is run when the thread group is executed. The identifier of a thread is equal to its coordinates in this system, with the thread group identifier added as an offset.

4.4.3 Dispatching a Compute Shader

When dispatching a compute shader, the number of thread groups for each dimension has to be passed along. To fill a 3D texture with a resolution of 256x256x256 pixels, a compute shader with thread groups the size of 4x4x4 threads needs to be dispatched with $256/4 = 64$ thread groups in each dimension.
Each thread group will be assigned its own identifier, the *thread group ID*.

4.4.4 Thread and Thread Group Identifiers

The thread group ID, or simply group ID, is calculated by taking the current index of the thread group in each dimension. When dispatching a kernel with 64 groups in each dimension, it that means that the group identifiers range from (0,0,0) to (63,63,63).
In each thread group, the thread ID is then calculated as follows. Given id_{group} is the thread group ID, $n_{threads}$ is the number of threads specified in the kernel, and t_{coord} are the coordinates of the thread inside the thread group, then:

$$id_{thread} = id_{group} * n_{threads} + t_{coord}$$

Here are two practical examples.

$$\begin{aligned} id_1 &= \ (0,0,0) * (4,4,4) + (0,0,0) = (0,0,0) \\ id_2 &= (1,63,2) * (4,4,4) + (3,3,1) = (7,255,9) \end{aligned}$$

The following figures each show a single thread group, with a specific thread marked in red. The group ID is denoted in the top left corner, together with the coordinates of the thread inside the thread group.

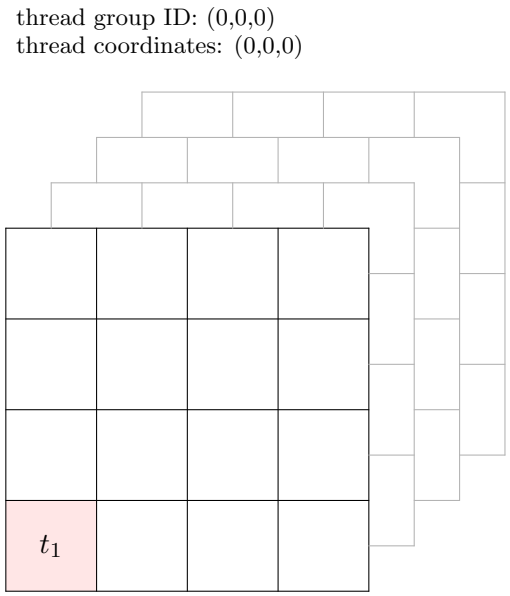


Figure 46: Compute shader thread group with dimensions 4x4x4. Marked in red is thread t_1 with $id_1 = (0,0,0)$.

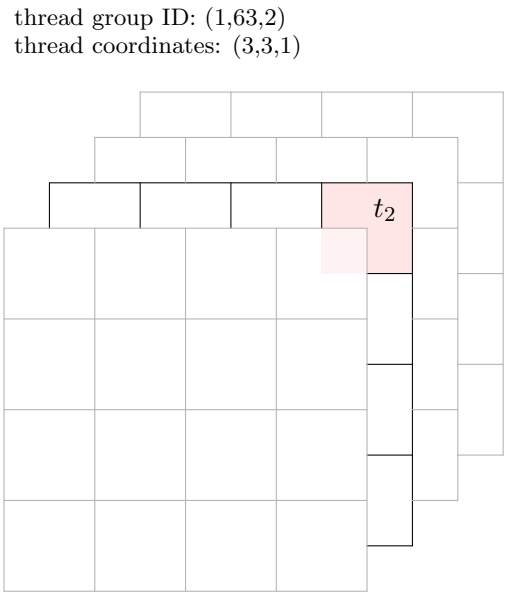


Figure 47: Compute shader thread group with dimensions 4x4x4. Marked in red is thread t_2 with $id_2 = (7,255,9)$.

4.4.5 Making Use of All Channels

Once it is clear how the dispatch thread ID is calculated, a compute shader can be implemented. The following example shows a kernel that computes a 3D Voronoi noise texture with the use of fractal Brownian motion, here called `fbm`. Each channel of the texture is assigned a noise value that was calculated with a different scale and a different amount of octaves.

```
[numthreads(8,8,8)]
void CSNoise (uint3 id : SV_DispatchThreadID)
{
    float3 threadCoord = float3(id.x, id.y, id.z) / 256;
    float3 noiseCoord = threadCoord * _NoiseScale;

    float r = fbm(noiseCoord * 1, 1);
    float g = fbm(noiseCoord * 2, 4);
    float b = fbm(noiseCoord * 5, 8);
    float a = 1;

    _Result[id.xyz] = float4(r, g, b, a);
}
```

Listing 7: An implementation of a 3D noise compute shader.

With each channel holding a different level of detail, all shaders that read from this texture can fetch multiple noise values with a single texture lookup. Listing 8 shows an excerpt of a standard fragment shader that reads from a 3D noise texture as described above.

```
float3 sampleDensity(float3 p) {
    return tex3D(_NoiseTexture3D, p).xyz;
}

fixed4 frag(v2f i) : SV_Target
{
    fixed4 color = 0;

    float3 samplePos = i.worldPos * _NoiseScale;
    float3 noise = sampleDensity(samplePos);

    float detail0 = noise.x;
    float detail1 = noise.y;
    float detail2 = noise.z;

    // making use of the different detail levels...

    return color;
}
```

Listing 8: An implementation of a shader making use of a 3D noise texture.

5 Technical Implementation

This section describes the current solution, the implemented software architecture and the final results of the project.

5.1 System Overview

The following graphics displays the current system overview. As described in subsection 5.3 and subsubsection 8.2.1, instead of the elevation model data from *swisstopo*, a Unity plugin for *ArcGIS* maps services was used. This change is reflected in the new system overview, and is the only adjustment that was made.

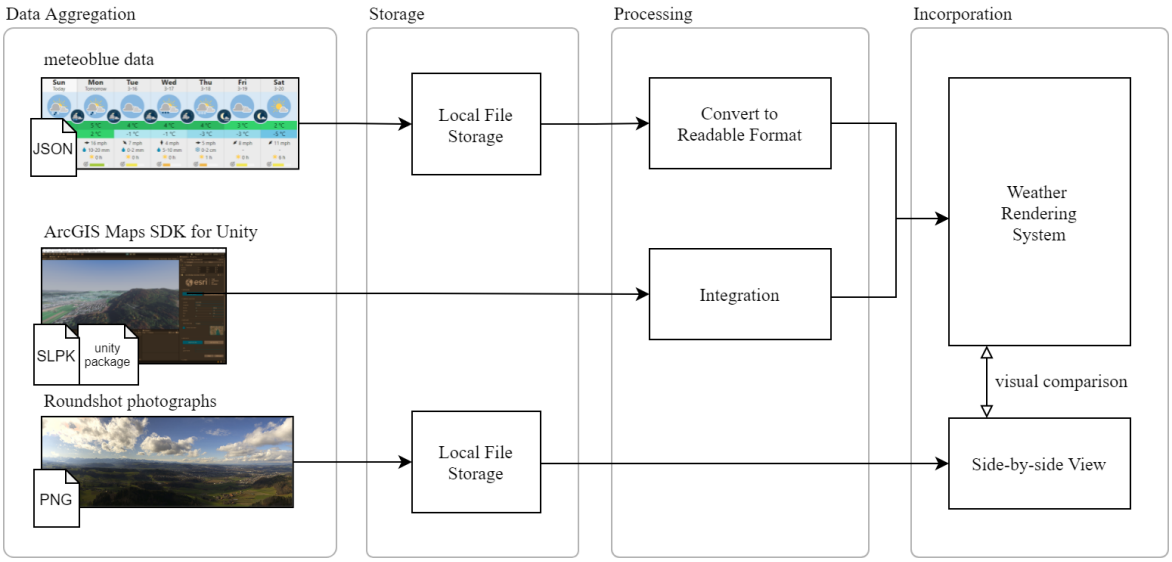


Figure 48: The system overview diagram.

5.2 Meteoblue Integration

While the bachelor project specification document described the use of *meteoblue*'s "basic 1h" package, this was not the only package that was used in the final implementation. After the cancellation of the technical interview with *meteoblue* and further research into additional, available weather data sources, the package "clouds 1h" was suggested. It provides an overview of cloud coverage percentage for each cloud layer, which is exactly what the implementation was missing.

Package name	**Description**
Basic (1h)	Mainly used for shading and visual effects.
Clouds (1h)	Mainly used for cloud coverage data.

The following table displays a list of properties from both data packages and how they are used in the final implementation.

Property name	**Source**	**Usage**
Wind speed	Basic (1h)	Used with a multiplier
Wind direction	Basic (1h)	Converted form degrees to a directional vector
Precipitation	Basic (1h)	Used for controlling rain particle system and cloud color. Factored into cloud edge highlight colors from sunshine. Factored into skybox colors.
Precipitation probability	Basic (1h)	Included in approximation of precipitation
Cloud coverage low	Clouds (1h)	Used for the weather in Bern
Cloud coverage mid	Clouds (1h)	Used for the weather in Bern
Cloud coverage high	Clouds (1h)	Used for the weather in Bern
Total cloud coverage	Clouds (1h)	Used for the distant weather

Other properties like temperature and UV index provide insufficient or irrelevant information and have therefore not been mapped.

5.3 ArcGIS Integration

During the research phase of the project, the *ArcGIS Maps SDK for Unity* [24] was found. This proved to be a suitable replacement for the originally planned *swisstopo* elevation model data. The plugin was easily installed. The setup process required minimal amount of effort and the plugin was ready to run in no time.

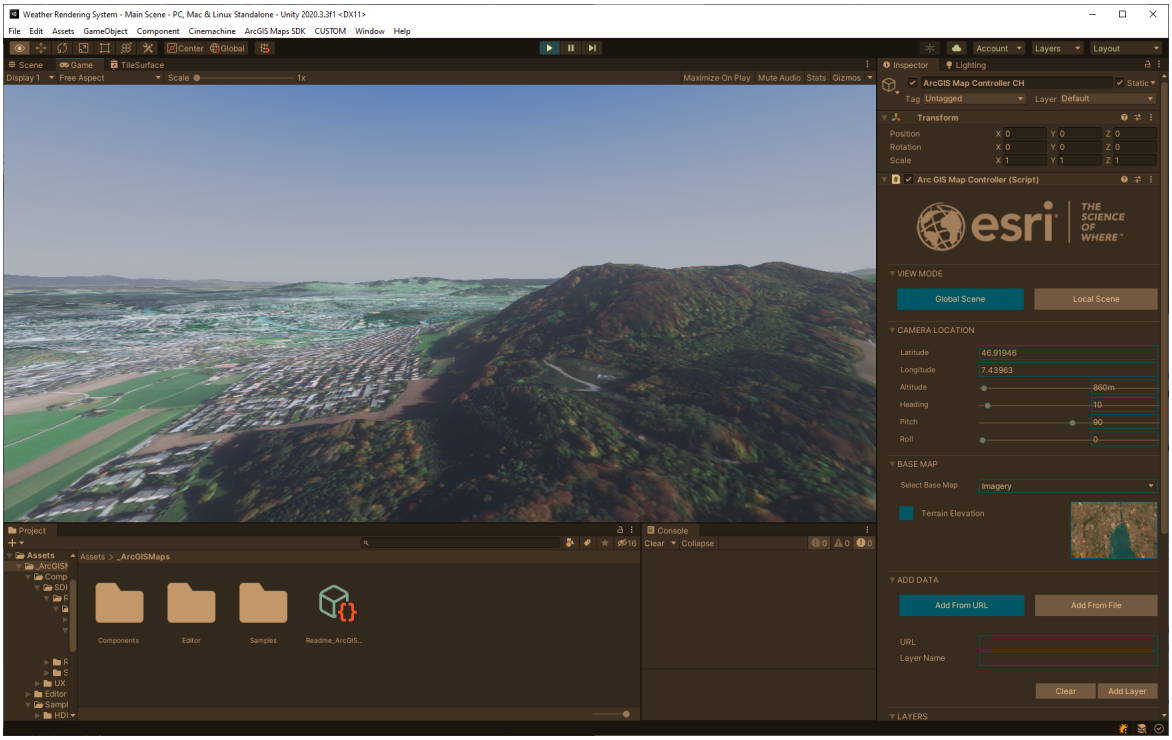


Figure 49: ArcGIS Maps SDK for Unity [24].

The plugin generates and renders map tiles according to the elevation model and automatically maps aerial photographs as textures on top. This happens during runtime and is continuously updated.
After setting up the plugin in Unity, the camera needed to be positioned on top of the Gurten mountain. Also, there needed to be a translation mechanic to move the camera to the top of the Bantiger mountain. This was done with via script and is not part of the *ArcGIS* plugin.

5.4 Unity Project Architecture

The project architecture in Unity consists of a set of volumetric shaders, the *ArcGIS* component and some auxiliary objects.
Figure 50 shows the content of the Unity scene file. The first section, named "general", contains the standard game objects for the primary camera and for the directional lighting.
The *ArcGIS* component only requires a single game object, which has been placed into its own section.
It is followed by the core of the weather rendering system, which is housed under the section named "clouds". In it, there are several controlling game objects like the "meteoblue master", which are responsible for setting up the shaders.
At last, the "UI" section contains the default game objects for a user interface (UI) in Unity, adjusted to this project.

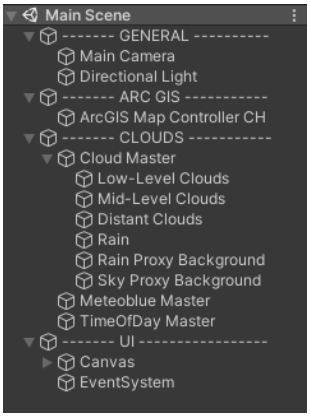


Figure 50: Hierarchy of the Unity project.

5.4.1 Scene Anatomy

The scene setup is similar to what was originally planned in subsection 3.2. There are multiple layers of clouds, each rendered by a unique volumetric shader. However, there is no layer for high-level clouds. It turned out that the intricate visual appearance of cirrus clouds was more difficult to simulate than anticipated. Also, there is no dedicated layer for ground fog, as the Unity engine already offers a way to include fog into the scene.
The current scene anatomy is depicted in the following graphic, as viewed from the side.

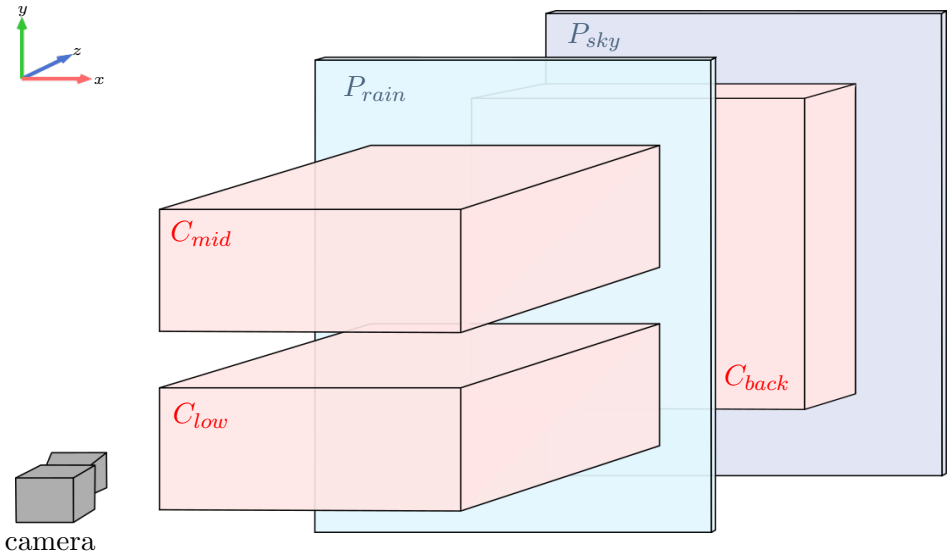


Figure 51: Final Unity scene anatomy.

In the scene, there are three cloud layers: C_{low}, C_{mid} and C_{back}. The shader for layer C_{low} handles low-level clouds like the puffy cumulus, while C_{mid} is responsible for rendering mid-level clouds of the "alto"-type. Both front layers use weather data for Bern, Switzerland. Behind the two front layers, there is C_{back}, which renders an approximation of cumulonimbus clouds. It uses the weather data of the correspondent location in the distance (either Solothurn or Fribourg) instead of Bern.

There are also two proxy objects, which are game objects that substitute a certain visual effect, like rain or the sun's halo. They are transparent planes and are placed in-between the cloud layers.

The proxy object P_{rain} is responsible for darkening the scene whenever it rains. The same effect could be achieved with post-processing, but using a proxy object offers finer and more customizable control over the effect.

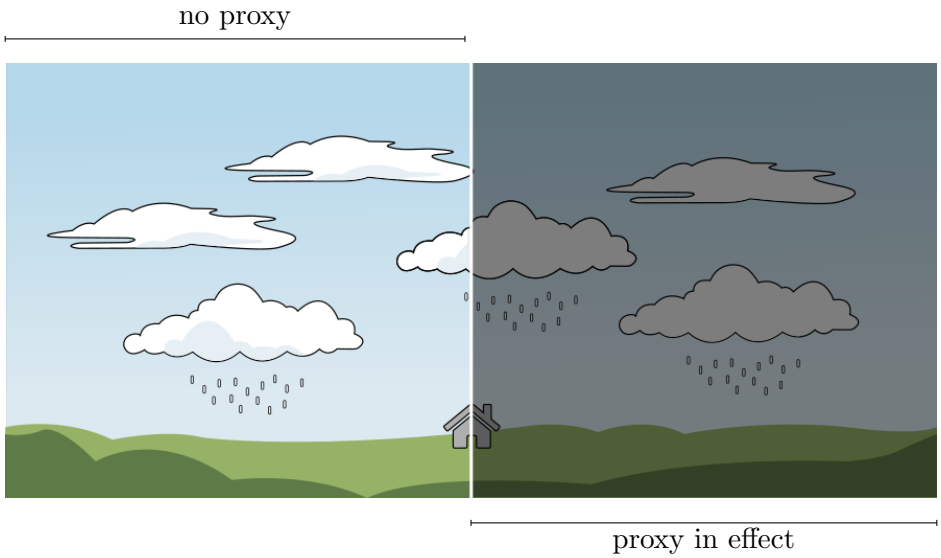


Figure 52: Desired effect of the rain proxy plane.

The other proxy object, P_{sky}, makes sure the sun is surrounded by a bright circle of the same color, supporting the strength of the sunlight in the sky. This effect is most prominent when the sun is setting or rising, giving the scene a more vibrant and natural look.

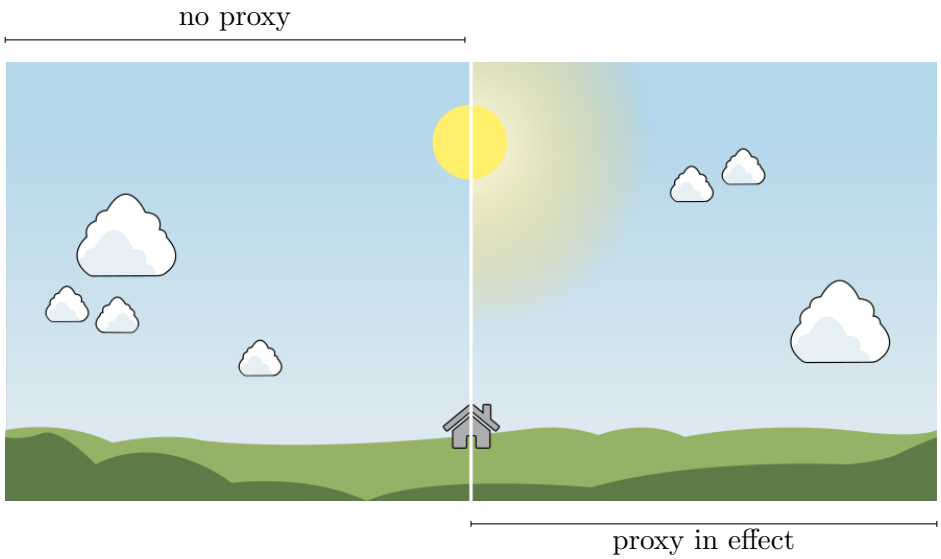


Figure 53: Desired effect of the sky proxy plane.

This effect could also be implemented in the skybox, but with the skybox of the high definition render pipeline (HDRP) being part of the post-processing, it is easier to maintain a proxy object than to create a custom post-processing effect.

5.5 Render Process

Each component in the scene fulfills a specific role. The following graphic lists the tasks that each component has to do during the start of the simulation (OnStart), and during each update loop (OnUpdate).

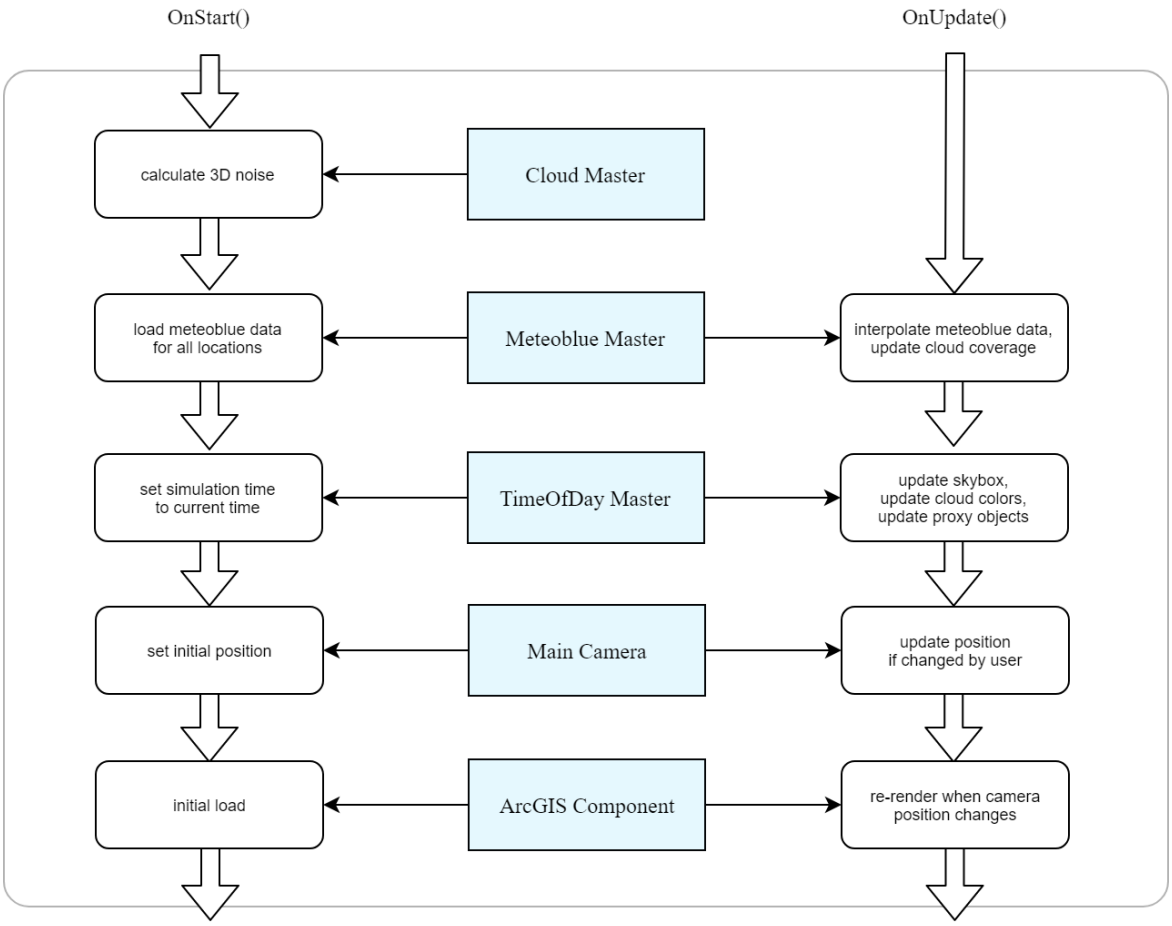


Figure 54: Render process of the Unity project implementation.

Since the 3D noise texture, that is generated at the start by the "cloud master", is seamless, it does not have to be recalculated every frame. After the *meteoblue* data is loaded by the "meteoblue master", the setup is almost complete. The default values for time of day and camera position are set while the *ArcGIS* component starts itself and runs automatically.

During the update loop of the engine, the weather data is interpolated from hour to hour, depending on the current time. This is necessary as the *meteoblue* data is only available for every full hour. This pseudo-code snippet shows how an average value for the weather data would be determined based on the time.

```
simulationTime = "18:45" // current simulation time (24h-clock)
time1 = floor(simulationTime) // "18:00"
time2 = ceil(simulationTime) // "19:00"
x = lerp(time1, time2, simulationTime) // 0.75

weatherData = lerp(weatherDataFor(time1), weatherDataFor(time2), x)
```

Listing 9: Pseudo-code for linear interpolation of weather data.

5.6 Noise Generation

As extensively described in section 4, the noise texture is generated by a compute shader. The resulting 3D texture is seamless and is solely based on the Voronoi noise generation algorithm, invoked with different scales and octaves. In the output texture, each texel holds four different noise values, one for each color channel.
The compute shader is dispatched only once, at the start of the simulation. This saves on a great deal of performance, as the noise does not need to be recalculated each frame.

5.7 Rendering Techniques

As mentioned in subsection 4.1, algorithms that have been documented in the previous project will not be described again. The current solution uses techniques like ray marching, light marching and sunlight forward scattering in a very similar fashion.

5.8 Shadow Casting

The volumetric cloud shaders do not contain a shadow pass, which means that the cloud objects do not cast shadows themselves. This is associated with the *ArcGIS* component.
The *ArcGIS* plugin generates geometry with an extremely high scale, where one in-engine unit corresponds to one meter in real life. The resulting geometry is huge compared to the rest of the scene. This is a problem, because the map's dimension exceed the capabilities and size of Unity's shadow texture. Unfortunately the geometry cannot be reduced in size so easily, because when reducing the scale of the generated map, the textures lose a good amount of detail.
However, the problem was solved with an alternative approach to casting shadows.

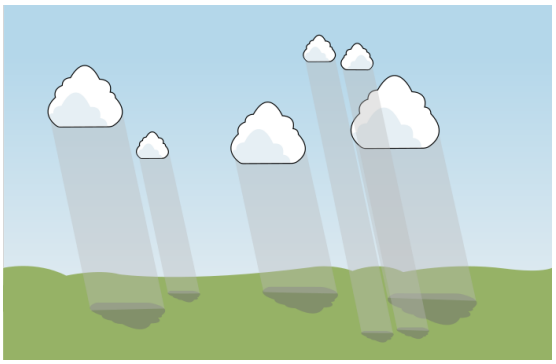


Figure 55: Shadow casting: Normally, objects would cast their own shadow in a dedicated shadow pass.

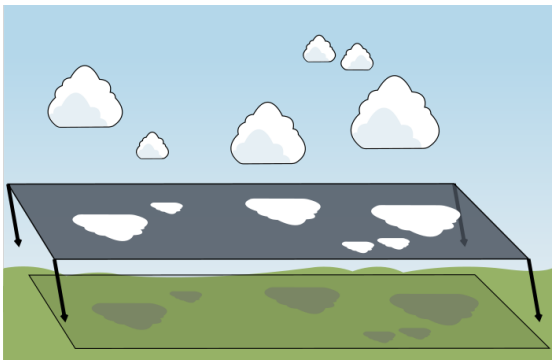


Figure 56: Shadow casting substitution: The 3D noise texture is sampled again in the ground shader and added as shadow.

What Figure 56 shows is not a plane that casts the shadow, but instead an illustration of how the data is used from the same noise texture that was used for the clouds. Thus, the shader for the ground tile, which is provided by the *ArcGIS* plugin, was modified to read the 3D noise texture. With that information, the ground shader knows where the clouds will be in the sky and adds a darkened, transparent layer over the existing ground color.

This method works well for the current solution, but would there be other objects than clouds, they would not cast shadows for the same reasons.

5.9 Results

The final implementation yields great results, visualizing the *meteoblue* weather reports in a variety of different settings, at all times of day.

5.10 Cloud Layers

As displayed in Figure 57, each cloud layer renders an approximation of the clouds it represents. In the low-level layer, there are small puffy cloudlets. In the alto-layer, there are dense fields of clouds and lastly, the distant cloud layer shows high-towering heaps of cumulonimbus approximations.

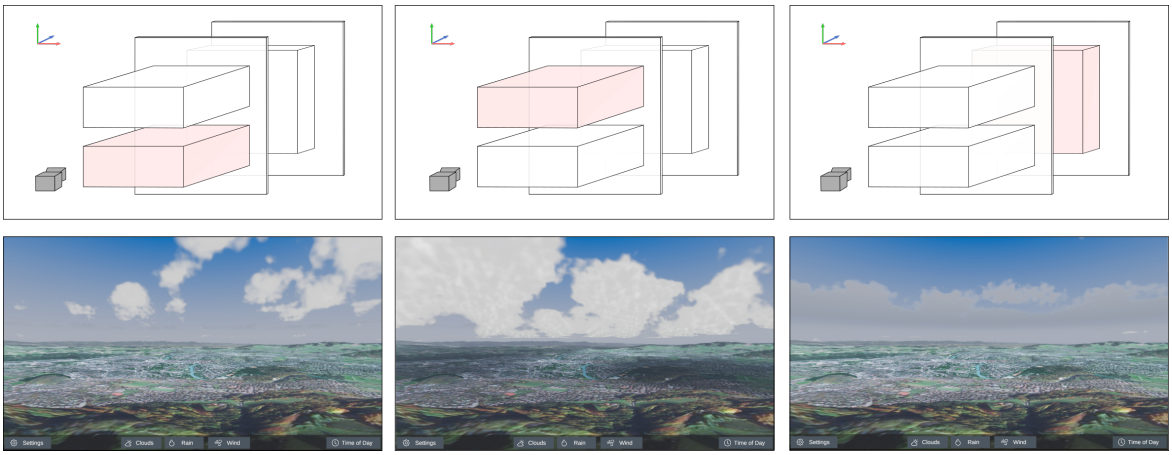


Figure 57: Side-by-side comparison of the cloud layers and their visual outputs.

Combined, the final output shows a diverse scene for the weather conditions extracted from the *meteoblue* data.

Figure 58: Final render output of the weather rendering system.

5.10.1 Times Of Day

The current solution is able to render the weather for all times of day. The following figures demonstrate the capabilities of the weather rendering system.

Figure 59: Final render output for a partly cloudy morning before sunrise.

Figure 60: Final render output for a early afternoon.

Figure 61: Final render output for a rainy evening.

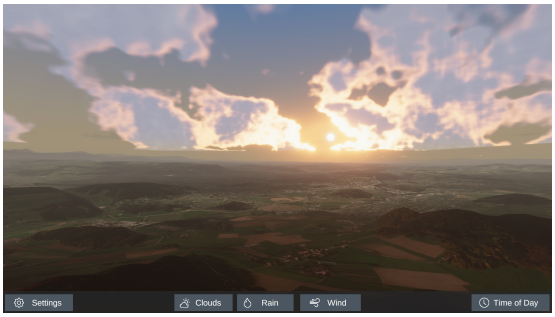


Figure 62: Final render output for a golden sunset.

Figure 63: Final render output for a evening right after sunset.

Figure 64: Final render output for a cloudy morning.

the colors of the clouds are influenced by the skybox and the sun's light. They also adapt to the amount of precipitation as well as the fog color. The clouds have an outer and inner primary color, both enriched with details from the noise texture. The outer edge of the clouds reacts the most to the sun's position and intensity, especially improving the scene at sunrise and sunset.

5.10.2 Proxy Objects

The proxy objects were used instead of complex, custom post-processing effects. Their effects are key to the credibility of realism of the rendered scenes. As the following graphics show, the proxy planes add a vital visual upgrade to the scenery.

Figure 65: Visual comparison of a rendered scene with and without the rain proxy plane.

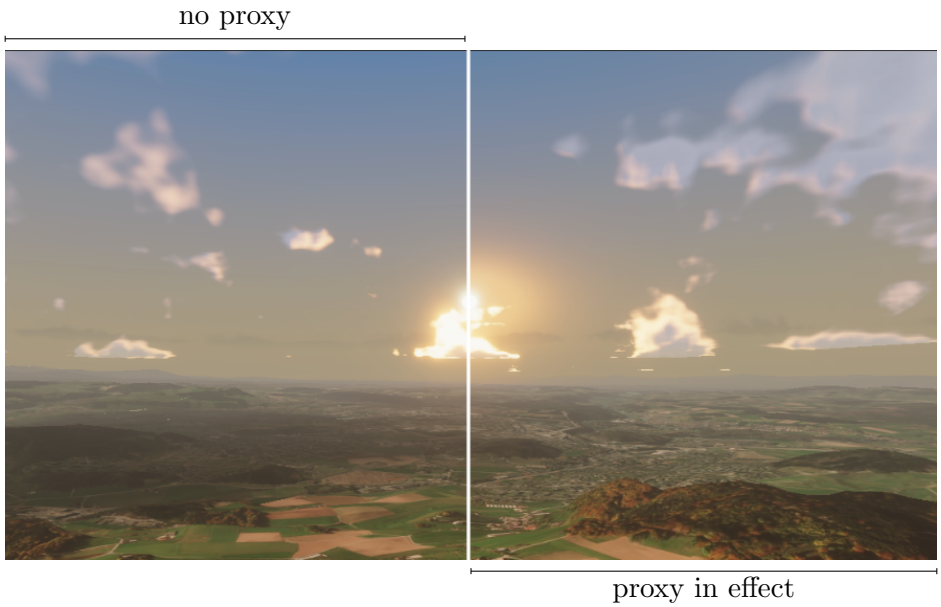


Figure 66: Visual comparison of a rendered scene with and without the sky proxy plane.

5.10.3 Shadow Mapping

As described in subsection 5.8, the cloud objects do not cast shadows. Instead, the terrain shader was modified to read the 3D noise texture and calculate the shadow itself .

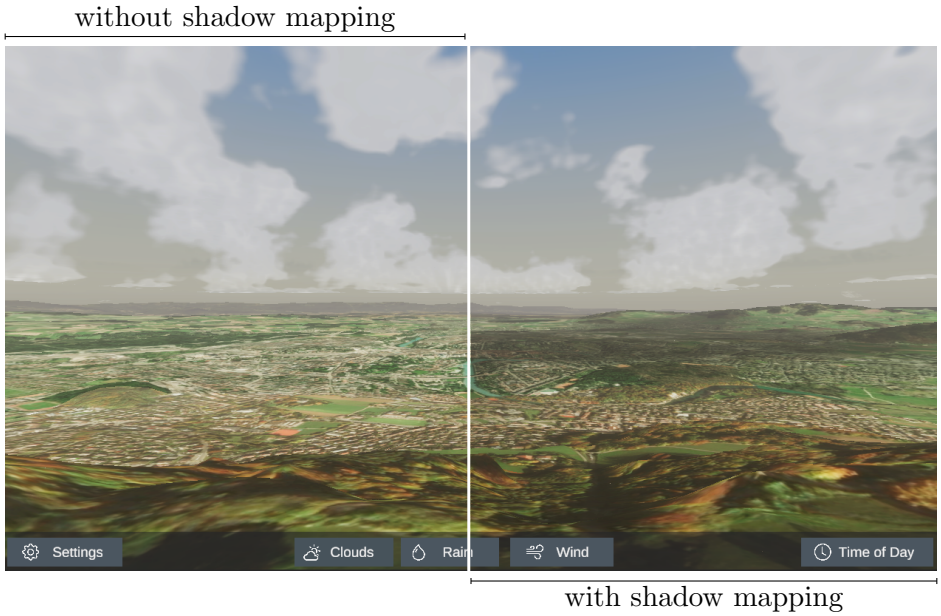


Figure 67: Visual comparison of a rendered scene with and without the shadow mapping on the ground tiles.

5.11 Visual Realism

Since this project renders cloudscapes based on real weather forecasts, it is only logical that the clouds are supposed to look as realistic as possible. To assess the realism of the rendered image, several aspects have to be considered.
First is the tone mapping of the image. The human eye is adapted to natural colors and quickly detects disparities in a color palette that does not reflect Nature's colors.
According to Rademacher [25], the shadow softness of cast shadows is a also great way of determining how realistic a rendered image looks. Apparently, the same is true for surface smoothness. Unfortunately, both these techniques are not applicable as the clouds have no smooth surface and the shadow is cast on a vibrantly colored terrain, making it hard to read where shadows start and end.
Rademacher makes another good point, which relates to the simplicity of the scene. He states that "[...] in a rendering application, it may be better to spend time on generating proper soft shadows and adequate textures, rather than adding more of the same lights or objects, or simply adding new objects for variety" [26].
This leaves to believe that in order to achieve higher fidelity and realism, the focus should be put on the texturing and shadow casting of the clouds, rather than the shape and quantity.

Still, there are some other methods involving neural networks that try to interpret the realism of a rendered image.

5.11.1 Convolutional Neural Network

Given there is a convolutional neural network (CNN) that is able to classify images of the sky, the weather or clouds into descriptive labels or even genera of cloud formations, then one could just seed those rendered images into the CNN and verify whether the results are truthfully showing "real" clouds. Of course, this is heavily dependent of how well the CNN was trained.

5.11.2 Generative Adversarial Network

A similar approach to the CNN is a generative adversarial network (GAN) setup. It describes two neural networks, which compete with each other in a cat-and-mouse game: The *generative* network tries to imitate the training set by generating artificial photographs with many realistic characteristics, while the *discriminative* network tries to tell whether the generated images are fake or not.
With this method, the rendered cloud images could be passed through the discriminative neural network to see if at least the network thinks the images are of real clouds.

5.11.3 Histogram Comparison

The *histogram* is a graphical representation of data like brightness or color distribution of a given photograph. When extracting the color histograms of the real photograph and the one of the rendered image, they could be compared and rated how different in color they are.

5.11.4 Professional Meteorological Assessment

Another viable solution is to let a professional meteorologist inspect and rate the rendered images and judge the realism of the depicted scenarios, which should reveal if the rendered clouds could actually form and exist in reality.

5.11.5 Measurability and Conclusion

The previous subsections suggest that there are ways to validate and interpret the visual realism of the rendered images, but no practical method to factually measure that value. As for this project, the suitable method to estimate the realism of a rendered image, that still conforms to the scope of the project, is comparing the in-engine render side-by-side with the *Roundshot* image.

5.12 Physical Accuracy

It is important to note that the project is only a visualization of weather forecasts, not a physically accurate simulator. Specifically the formation and dissipation of clouds, the sun's position in the sky, and the wind speed do not necessarily match their real counterparts.

5.13 Roundshot Image Overlay

The following graphic shows how the *Roundshot* images are integrated into the weather rendering system. When the application is running in "real-time mode", which is the mode that visualized the *meteoblue* data, then a button at the bottom of the UI lets the user toggle an overlay. The overlay shows a photograph of the same location and of the corresponding time and date, taken by a *Roundshot* camera.

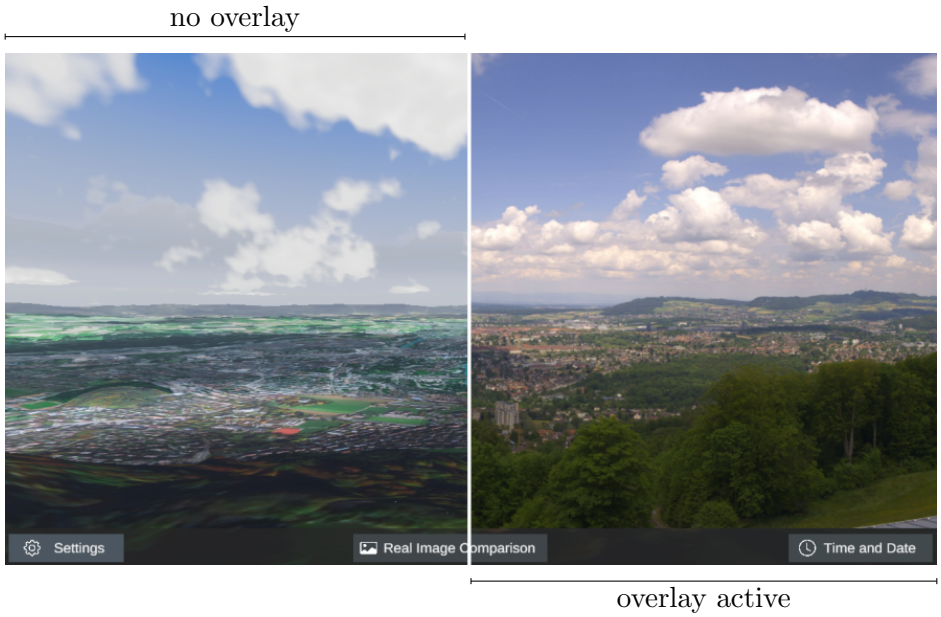


Figure 68: Visual comparison of the rendered scene with and without the *Roundshot* image overlay activated.

Figure 68 only shows a visual comparison of the two cases where the overlay is activated and where it is not. In the application, the overlay spans over the whole viewport instead of just half.
The comparison brings out a couple of things that are missing in the weather rendering system, but especially that there is room for improvement regarding the cloud shapes. Also, the cloud colors lack diversity and are too uniform to be naturally realistic.

5.14 Comparison to Previous Work

The prototype created in the previous work was able to render a single layer of clouds. There was no UI and the scene only contained mockup plants and rocks. That implementation was able to run at approximately 30 FPS in Full HD. The ray marching and step count was 25, while there were only two steps for the light marching. Therefore, the number of texture lookups $s_{previous}$ adds up to a total of:

$$s = n_{layers} * (steps_{ray} * steps_{light})$$
$$s_{previous} = 1 * (25 * 2) = 50$$

The new solution that was achieved in this project has three cloud layers instead of just one. It also uses 100 steps for ray marching and 25 for light marching.

$$s_{current} = 3 * (100 * 25) = 7500$$

Knowing that the current solution runs at 60 FPS, this gives an approximate performance gain of factor $(7500/50) * (60/30) = 300$. That is an astounding **3000%** more powerful.
For almost all of this, credit has to be given to the use of a compute shader. The fact that the noise has to be calculated only once instead of every frame allows for much higher fidelity in rendering the clouds.

Some of the code of the previous work was used as a template. However, a flaw in the ray marching algorithm was discovered. Fixing that also caused a noticeable increase in performance.

5.15 Comparison to Other Work

As it happens, Microsoft's Flight Simulator game (2020) also uses weather data from *meteoblue* to render its volumetric clouds [27]. The results are extraordinarily realistic, but also certainly based on an extensively more complex solution than simple data packages, like in this project.

Figure 69: Screenshot of Microsoft's Flight Simulator, flying over Bern.

Figure 70: Final render output of weather rendering system.

The comparison shows that the colors of the environment is just as important as the realism of the clouds. For example, the vibrant colors towards the horizon in Figure 69 give the scene a much more natural appearance.

6 Testing

The bachelor project specification document envisioned the following cases to be tested and validated. The tables each show the test case and the related results, and whether or not the test has been passed.

6.1 External Data Testing

6.1.1 Weather Data

Case	T.1
Test case	Weather data
Expected result	The data from *meteoblue* is incorporated into the weather rendering system. The data directly controls all related variables.
Actual result	✓ *meteoblue* data stored periodically ✓ Data read at start of system ✓ Data controls the simulation
Passed?	Yes

6.1.2 Terrain Data

Case	T.2
Test case	Terrain data
Expected result	The data from *swisstopo* is incorporated into the weather rendering system. The elevation model defines the terrain height map. The aerial images are used for texturing.
Actual result	× *swisstopo* data not used ✓ Substitution found for elevation model data ✓ *ArcGIS* plugin is in use and functional
Passed?	No / Substituted

6.1.3 Photographic Data

Case	T.3
Test case	Photographic data
Expected result	There is a feature that allows to overlay the *Roundshot* photograph of the same time and date as the rendered image was created for.
Actual result	✓ *Roundshot* images are stored locally ✓ The described feature is included in the UI
Passed?	Yes

6.2 Functional Testing

6.2.1 Code Functionality

Case	T.4
Test case	Code functionality
Expected result	The code for the weather rendering system compiles and runs without error.
Actual result	✓ The code compiles successfully ✓ The code runs error-free
Passed?	Yes

6.2.2 User Interface

Case	T.5
Test case	User interface
Expected result	The user is able to switch between the two modes, "real mode" and "play mode". The user is also able to control the weather system over the UI accordingly.
Actual result	✓ The user can switch between the two described modes ✓ The user can control the weather manually in "play mode" ✓ The user can choose the date and time in "real mode"
Passed?	Yes

6.2.3 Performance

Case	T.6
Test case	Performance
Expected result	The shader code should run with reasonably good performance and should not show visual stutters or frame drops.
Actual result	✓ The simulation runs at approximately 60 FPS in Full HD ✓ There are no noticeable frame drops ✓ There are no visual stutters
Passed?	Yes

6.3 Visual Testing

6.3.1 Real Photographs

Case	T.7
Test case	Real photographs
Expected result	The visual output of the weather rendering system is to be compared with live weather photographs from *Roundshot* cameras. The rendered image should resemble the weather of that time, to a reasonable extent.
Actual result	✓ The visual output resembles the weather for the same time to a reasonable extent × Clouds of the family "cirrus" are missing
Passed?	Partially

6.3.2 Similar Products

Case	T.8
Test case	Similar products
Expected result	The visual output of the weather rendering system is to be compared with the in-game footage of Microsoft's *Flight Simulator* game. The rendering system should achieve similar results, to a reasonable extent.
Actual result	✓ The visual output resembles that of Microsoft's Flight Simulator for similar weather conditions
Passed?	Yes

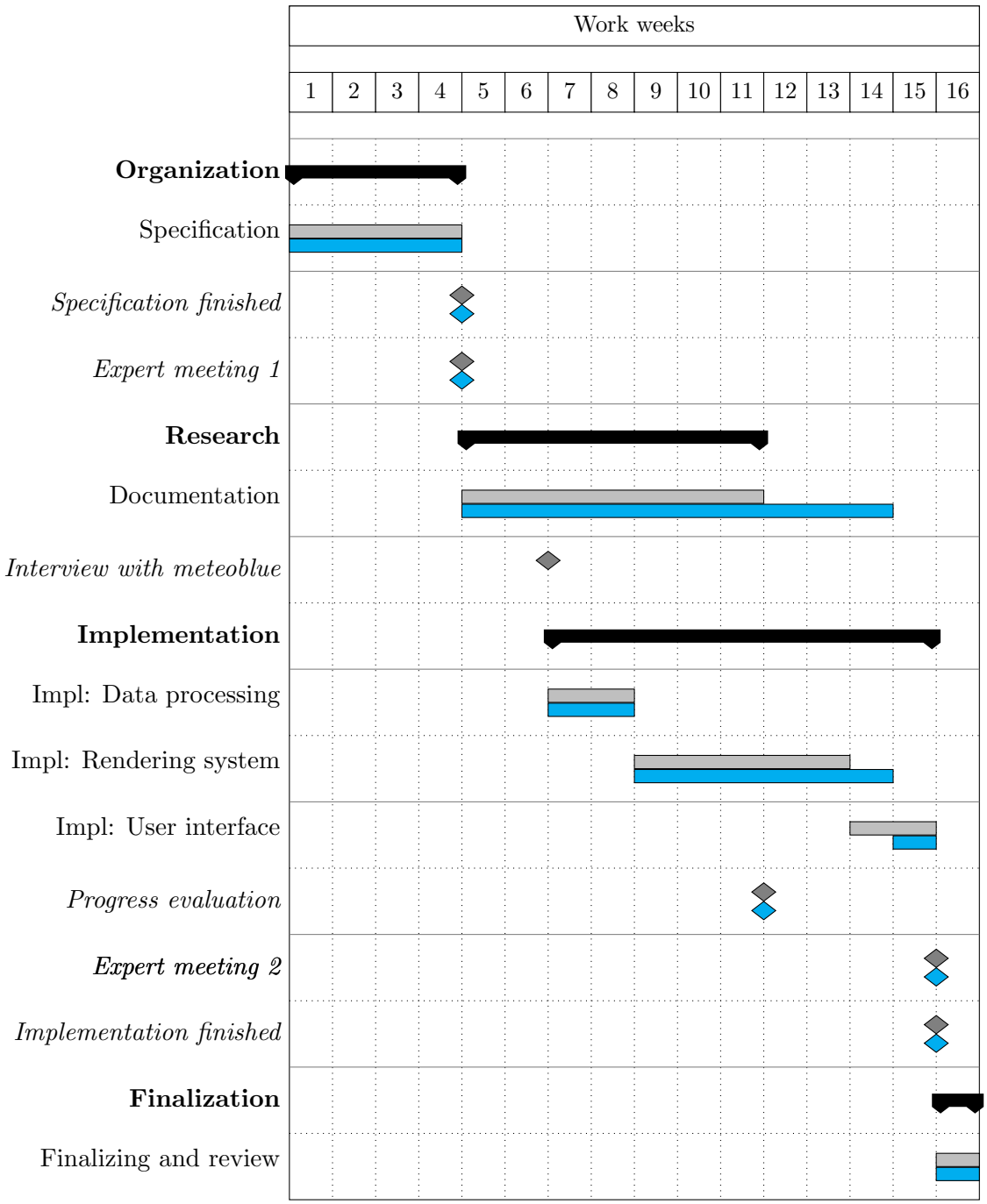
7 Conclusion and Critical Discussion

Viewed from a critical perspective, the project fulfilled or substituted all of its requirements and passed all its tests. The *meteoblue* data is incorporated into the system, as well as the elevation model from *ArcGIS* and the live photographs from *Roundshot*. In section 3, a prediction model was described where data of future forecast would be used to estimate the cloud types, but this was not implemented in the final weather rendering system, as this data was provided by the "clouds-1h" package.
The new system outperformed the previous prototype by a factor of 300. Thus, the use of a compute shader was definitely worth the effort. Compared to state-of-the-art cloud systems, the one made during this project can keep up in terms of customization options, but lacks some realistic details and the ability to render cirrus clouds.

8 Project Management

8.1 Schedule Comparison

The following chart shows the original schedule (in grey) with a side-by-side comparison with the actual time spent for each task (in blue). It indicates that the schedule was mostly met throughout the project.



The only major discrepancy between the planned and the actual schedule is the interview with *meteoblue*, which was turned down by *meteoblue*. The interview would overstep the

limited license that was arranged for this project. This was unfortunate, but left more time for the documentation, which came in handy.

8.2 Fulfillment of Requirements

The following table shows the original requirements and whether or not they were met during the project.

N°	Name	Is met?
R.1	Understanding the basic nature of clouds	Yes
R.2	Understanding of different characteristics of clouds	Yes
R.3	Understanding of compute shaders	Yes
D.1	Periodical acquirement of real-time weather data from *meteoblue*	Yes
D.2	Periodical acquirement of photographs of 360-degree cameras	Yes
D.3	Acquirement of elevation model data from *swisstopo*	Substituted
D.4	Noise generation based on compute shaders	Yes
D.5.1	Implementation of data aggregation and processing	Yes
D.5.2	Implementation of core rendering system	Yes
D.5.3	Implementation of user interface	Yes
O.1	Rendering performance optimization	Partially

Both requirements D.3 and O.1 are not or only partially met.

8.2.1 Elevation Model Replacement

Originally, it was planned to retrieve elevation model data from *swisstopo*. This turned out to be more problematic than anticipated, having multiple file formats that are incompatible with Unity Engine. This would have required a middleware to convert the files to a suitable format, which was also difficult to find. Additionally, the elevation model data was split into tiles, which would have needed to be stitched together in-engine.

Luckily, during research, a Unity Engine plugin for *ArcGIS* services developed by *ESRI* was found. This allowed for a quick setup without further mapping and tiling of geodata or aerial images.
Since the generated 3D models from the plugin were more than a suitable replacement for the *swisstopo* data, the requirement D.3 was annulled, as there was now another, easier way to get 3D elevation data. From then on, only the *ArcGIS* plugin was used.

8.2.2 Rendering Performance Optimization

The use of a compute shader resulted in a massive performance boost. There have been several other attempts at additionally optimizing the rendering performance. This includes the reduction of ray marching and light marching steps, the careful choice of scale and amount of octaves for 3D noise generation as well as the avoidance of using heavy operations like `pow()`, `exp()`, `log()`. However, there was not sufficient time to delve deeper into performance optimization, but since the software is able to run with a decent frame rate, the issue was no longer pursued.

8.3 Future Work

8.3.1 Rendering Capabilities

Despite the considerable effort already put into lighting and illumination methods, there are still some features missing. One of those are god rays, the volumetric light shafts that shine through gaps in clouds, giving the scene even more depth. Other absent features are the sun's and moon's halo: A bright circle around the celestial body.

8.3.2 Live Data Feed

The resulting weather rendering system is still dependent on locally accumulated meteorological data. A potential follow-up project could be an implementation of a live data feed system that continuously updated the weather rendering system.
Technically, this would already be possible for the current solution. The question is rather if such a data stream exists and what license would be required to use it.

8.3.3 Simulation Game

Alternatively, the project could be turned into a simulator game, similar to how Microsoft's Flight Simulator works.

8.3.4 Meteorological Events

There is also the option to include meteorological events like thunderstorms, blizzards, rainbows and many more natural phenomena.

8.4 Project Conclusion

It is noteworthy that three more weeks were put into research. This was to be expected, because new algorithms have been continuously researched during implementation, which had to be documented. However, this did not conflict with the remainder of the schedule.
Also, the implementation of the rendering system itself took one more week than planned. This is negligible, though, as the implementation of the UI was completed quicker than anticipated.
Still, the total amount of time spent was about ten percent more than the originally estimated time budget of 320 hours. This is probably due to the fact that there was quite some effort put into the final implementation and graphical illustrations for the documentation.
The project occurred during the same time as the global coronavirus lockdown restrictions, but was unhindered by that.

In summary, all mandatory project requirements were met or substituted, almost all milestones were completed in time and the final implementation turned out great, making this a successful and very informative project.

9 Declaration of Primary Authorship

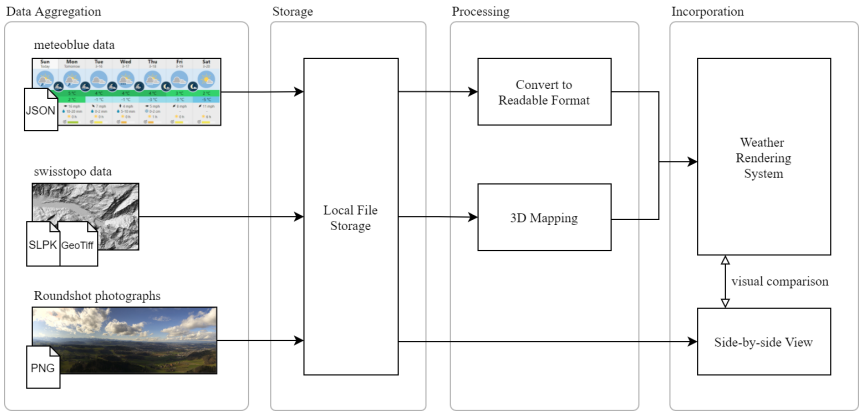
I hereby confirm that I have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by me are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Bern, June 10, 2021
Last Name, First Name:	Matthias Thomann
Signature:	.....

A Bachelor Project Specification Document

Real-time Weather Rendering System

Requirements Specification



Field of Studies:	BSc in Computer Science
Specialization:	Computer perception and virtual reality
Author:	Matthias Thomann
Supervisor:	Prof. Urs Künzler
Date:	April 14, 2021
Version:	1.0

Contents

1 General

1.1 Purpose

This document serves the purpose of defining and clarifying the goals, which the thesis 'Real-time Weather Rendering System' is supposed to achieve. Furthermore, the requirements specification allows for a more accurate evaluation of the achievement of objectives and of the result itself.

1.2 Revision History

Version	Date	Name	Comment
0.1	February 27, 2021	Matthias Thomann	Initial draft
0.2	March 03, 2021	Matthias Thomann	Updated project schedule
0.3	March 11, 2021	Matthias Thomann	Determined requirements
0.4	March 13, 2021	Matthias Thomann	Reworked vision chapter
0.5	March 14, 2021	Matthias Thomann	Added system overview diagram
0.6	March 15, 2021	Matthias Thomann	Added UI mockups
0.7	March 17, 2021	Matthias Thomann	Review update
0.8	March 18, 2021	Matthias Thomann	Updated system overview diagram
0.9	March 19, 2021	Matthias Thomann	Reworked project schedule
1.0	March 19, 2021	Matthias Thomann	Finalized document

2 Vision

2.1 Weather Rendering System

This section defines a high-level vision of the desired outcome of this thesis and potential future work. As listed in the primary goals, the weather rendering system will be making use of compute shaders. Compared to the prototype from the previous project, this is expected to result in a much better performance. That in turn, allows for a more complex and realistic model.
With the incorporation of real-time weather data and the use of topological landscape data, any given weather scenario for any specific location could be simulated and rendered. The desired outcome ideally looks similar to the image depicted in Figure 1. A rendered version of such a cloud system can look elusively realistic compared to an actual photograph, like in Figure 2.

Figure 1: A rendered image of volumetric clouds [1].

Figure 2: A photographic reference of clouds [2].

The first thought about the practical use of a fully-fledged volumetric cloud system might be a video game, since clouds are often a significant part of outdoor scenery in games. However, for this thesis it is intended that the knowledge and results acquired during the given period will be used to recreate a lifelike weather system instead.

2.2 System Overview

To get a better understanding of how such a system could be implemented, this diagram shows all the involved components and their processes.

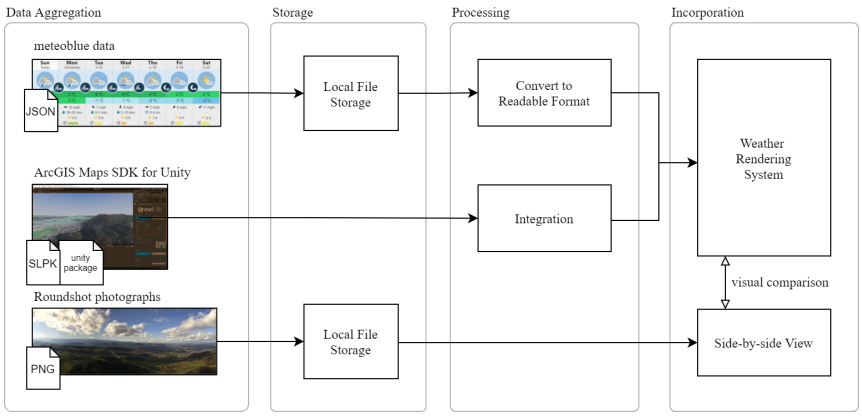


Figure 3: System overview diagram.

All external data is retrieved regularly and stored on the local file system. The file format of each data source is denoted in their respective bottom left corner. After aggregation, the data is processed into a readable and compatible format for the Unity Engine. From there, the weather rendering system can make unrestricted use of the data. The resulting output of the system can then be compared side-by-side with the collected live weather photographs.

2.3 External Data

2.3.1 Meteoblue Weather Data

To accurately model a weather system, conditions like precipitation, wind and cloudiness will be considered. Fortunately, the company *meteoblue* offers this data in form of different data packages [3]. As an additional bonus, the license costs are drastically reduced for student projects and educational work.
From all available data packages, the "basic_1h" [4] offer seems the most fitting for this thesis. It includes the most common weather variables only, but this will be sufficient for the planned project. Some of the crucial variables from that package are wind speed, wind direction, temperature, sea level pressure, and a pictocode (numeric value indicating the estimated cloudiness of the sky).
The weather data will be requested for the following locations:

- Bern, Switzerland
- Fribourg, Switzerland
  (to account for the weather in the background of the photographs)
- Solothurn, Switzerland
  (to account for the weather in the background of the photographs)

This data is retrieved on a daily basis and stored on a local file system for the duration of the thesis. The file format is Java-Script object notation (JSON).

2.3.2 Roundshot Photographs

The weather data from *meteoblue* gives detailed information about the weather at a specific time and date. But to be able to compare the rendered result of the weather system with the actual weather of that period, real photographs of the same time should be used. For that purpose, images taken by the *Roundshot* camera system from the company *Seitz* [5] are stored periodically.
There are many installations of those systems across the country. For this project, the following two locations are used:

- Roundshot camera Bantiger, Switzerland [6]
- Roundshot camera Gurtenpark, Switzerland [7]

This data is retrieved on a weekly basis and stored on a local file system for the duration of the thesis. The file format is portable network graphic (PNG).

2.3.3 Swisstopo Elevation Models

The last part of a convincing weather rendering system is the landscape, for which the topologically accurate 3D elevation model data from the federal office *swisstopo* will be used [8]. As of March 2021, *swisstopo*'s elevation models and landscape data are available free of charge [9]. The goal is to download and convert this data into a Unity-compatible 3D model and use it as a base for the scenery.
This data is retrieved once or whenever an update is due and stored on a local file system for the duration of the thesis. The file format is ESRI scene layer package (SLPK) or any other suitable data format made by the company ESRI.

2.4 UI Mockup

As for the user interface (UI), there will be two possible modes to choose from. The first one is a "live" mode that lets the user choose time and date, for which the weather is then recreated with the *meteoblue* data from that period. The other mode will be a "play" mode, where all influential weather variables can be controlled manually by the user.
The following mockups only serve as a general guideline and are not final.

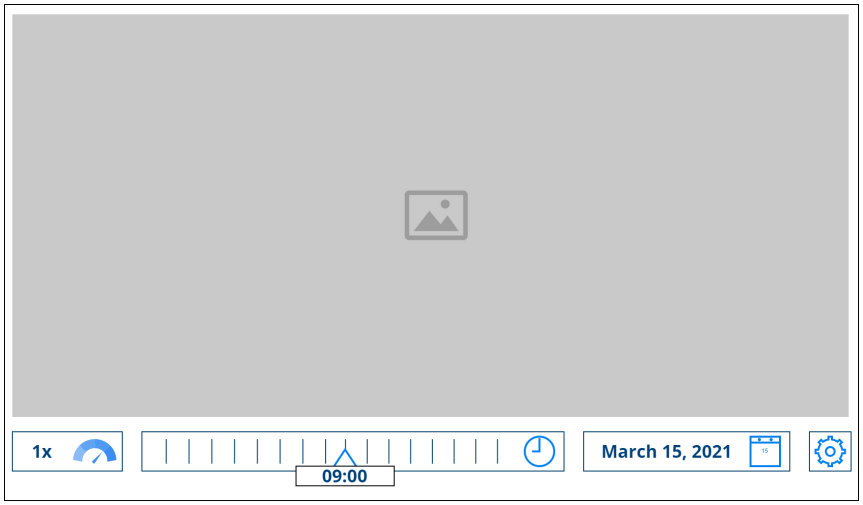


Figure 4: UI mockup of the "live" mode.

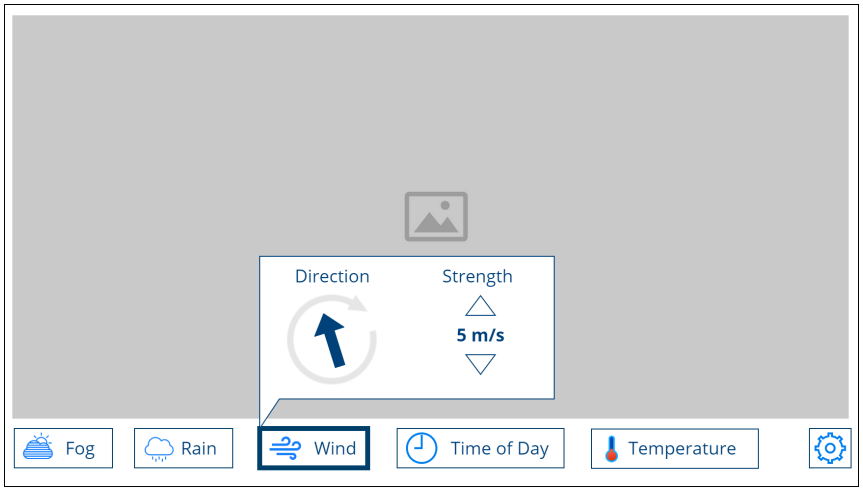


Figure 5: UI mockup of the "play" mode.

3 Scope of Work

3.1 Initial Situation

In computer graphics, especially in games, some features recur in an astonishingly large amount of platforms and genres. With the most obvious ones being water surfaces, cloudscapes and fire effects, they are present in almost any game. Naturally, those features grew in complexity, customizability and computational demands over time.
One of the core mechanics of those features is called a *volumetric shader*. A prototype of such a shader has been created in a previous project and will be used as base.

3.1.1 Previous Work

In a previous project, the process of creating a volumetric shader has already been researched and implemented in a prototype. Thanks to its high flexibility, different cloudscapes could be rendered by the same shader.

Figure 6: Result of the previous work's shader (Evening setting).

Figure 7: Result of the previous work's shader (Day setting).

During that project, some other important topics have been studied. Among those were volumetric rendering, Perlin and Voronoi noise generation algorithms, and a technique called *ray marching*.
The implementations of those algorithms and methods will most likely be reused in this thesis and will be adapted and improved accordingly.

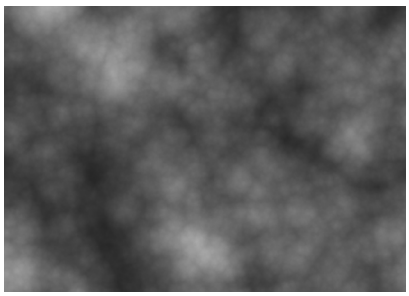


Figure 8: A generated noise texture with Voronoi's algorithm.

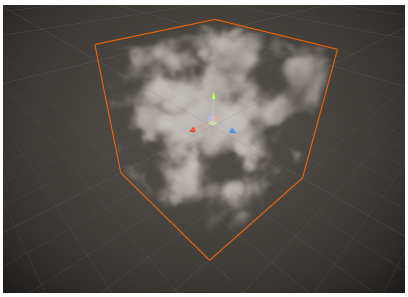


Figure 9: A screen capture of an image rendered with ray marching.

3.1.2 Near Real-time

As mentioned in subsection 2.3, the data will not be retrieved in real-time, but rather be polled periodically. However, when weather data is requested, the data for the current day as well as a seven-day forecast is returned. This means that the data, even though not obtained in real-time, can be processed for the real current time with the aid of interpolation.

3.2 Goals

As the title of the thesis suggests, this work will primarily focus on clouds and cloudscapes. The primary goal of the project is to research and implement rendering techniques for a real-time procedural weather rendering system. Another important goal is the incorporation of external data to achieve as much realism as possible.
The goals will be split into two distinct groups: mandatory and optional. However, this section only defines high-level goals. A detailed specification of all requirements can be found in section 4.

3.2.1 Mandatory Goals

The following tasks must be accomplished during the project:

- Understanding the basic nature of clouds
- Understanding of different characteristics of clouds
- Understanding of compute shaders
- Implement a weather rendering system
- Incorporate real-time weather data from *meteoblue*
- Incorporate topological landscape models from *swisstopo*

3.2.2 Optional Goals

For further optional work, these tasks can be looked into:

- Rendering performance optimization
- Automatic validation of realism of rendered cloudscapes
- Automatic comparison of rendered cloudscapes and photographs
- Automatic categorization of rendered cloudscapes

Most of the tasks labelled with "automatic" could be solved using a neural network. Since diving into the vast world of neural networks would go beyond the scope of this work, those goals are not considered compulsory.

3.3 Educational Objectives

Educational objectives include shader programming, knowledge about compute shaders, rendering techniques, common algorithms used in computer graphics like noise generation, a general understanding of aspects needed to create a complete weather system and finally the incorporation of weather and landscape data from third parties.

3.4 Used Software and Tools

All documentation will be written in LaTeX with Visual Studio Code. The shader will be implemented in Unity. The chosen shader language is high-level shading language (HLSL). For the presentation, Microsoft PowerPoint will be used.

3.5 Available Hardware

For development, the author's personal computer will be used. However, should the need arise, the Berner Fachhochschule can provide a set of powerful computers to use for the thesis.

4 Requirements

All requirements are grouped by type. This results in two major groups, which are research and development requirements. Each one of the requirements is derived from a goal listed in subsection 3.2.

4.1 Research Requirements

Each research requirement is denoted with the letter "R" followed by its number.

Number	R.1
Name	Understanding the basic nature of clouds
Description	In order to be able to recreate a realistically looking cloud shape, one has to examine and understand the way a cloud forms and disperses again first.

Number	R.2
Name	Understanding of different characteristics of clouds
Description	Among other characteristics, altitude, humidity and atmospheric pressure dictate the look and genus of a cloud. The goal is to decide which cloud types are required for a believable weather system.

Number	R.3
Name	Understanding of compute shaders
Description	Compute shaders proved to be a highly efficient tool when it comes to heavy calculations, like simulations. To improve performance and therefore allow for a more sophisticated weather system, compute shaders have to be researched.

4.2 Development Requirements

Each development requirement is denoted with the letter "D" followed by its number.

Number	D.1
Name	Periodical acquirement of real-time weather data from *meteoblue*
Description	In order to achieve a high degree of realism, real-time weather data will be used. *Meteoblue* offers different data package contracts, of which the "basic_1h" is to be acquired. The data will be downloaded daily. The usage of the data package requires physical locations. The chosen locations are: • Bern, Switzerland • Fribourg, Switzerland • Solothurn, Switzerland

Number	D.2
Name	Periodical acquirement of photographs of 360-degree cameras
Description	A comparison of real-time weather data with an actual photographic reference from that date and time will prove to be useful. Images from such cameras will be stored periodically on a local file system. The chosen system is that of the company *Seitz* called *Roundshot*, with these locations: • Roundshot camera Bantiger, Switzerland • Roundshot camera Gurtenpark, Switzerland

Number	D.3
Name	Acquirement of elevation model data from *swisstopo*
Description	The 3D elevation model data from *swisstopo* will be downloaded and mapped into a compatible format for Unity. This is then used as a base for the scenery. For texture layers, aerial images from *swisstopo* will be used and mapped onto the 3D model.

Number	D.4
Name	Noise generation based on compute shaders
Description	To make full use of the power of compute shaders, it is best to let them execute computationally demanding tasks. In this case, specifically noise generation.

The implementation requirement D.5 is split into three sub-requirements, D.5.1, D.5.2 and D.5.3.

Number	D.5.1
Name	Implementation of data aggregation and processing
Description	The first part of the weather rendering system is the external data aggregation and processing.

Number	D.5.2
Name	Implementation of core rendering system
Description	The second part of the weather rendering system is the core of the system itself. This includes noise generation, volumetric rendering, terrain generation, and so on.

Number	D.5.3
Name	Implementation of user interface
Description	Finally, the user should be able to control the weather system with an intuitive UI. The user can also switch between "live" and "play" mode, as described in subsection 2.4.

4.3 Optional Requirements

Gathered from subsubsection 3.2.2, there is one optional goal.

Number	O.1
Name	Rendering performance optimization
Description	This includes optimizing shader code, finding early exits for looping algorithms and reducing the overall workload of processing the external data during runtime.

4.4 Summary of Requirements

The requirements are each prioritized with a number from one (1) to three (3), with 1 being the highest priority and 3 being the lowest priority.

Number	**Name**	**Priority**
R.1	Understanding the basic nature of clouds	1
R.2	Understanding of different characteristics of clouds	1
R.3	Understanding of compute shaders	1
D.1	Periodical acquirement of real-time weather data from *meteoblue*	2
D.2	Periodical acquirement of photographs of 360-degree cameras	2
D.3	Acquirement of elevation model data from *swisstopo*	2
D.4	Noise generation based on compute shaders	1
D.5.1	Implementation of data aggregation and processing	2
D.5.2	Implementation of core rendering system	1
D.5.3	Implementation of user interface	3
O.1	Rendering performance optimization	3

5 Testing

The project will be implemented and tested in Unity. For testing, the following test cases can be used to verify and evaluate the implementation. They are split into two groups, separating the incorporation of external data with the implementation of the system.

5.1 External data testing

Case	Test case	Expected result
T.1	Weather data	The data from *meteoblue* is incorporated into the weather rendering system. The data directly controls all related variables.
T.2	Terrain data	The data from *swisstopo* is incorporated into the weather rendering system. The elevation model defines the terrain height map. The aerial images are used for texturing.
T.3	Photographic data	There is a feature that allows to overlay the *Roundshot* photograph of the same time and date as the rendered image was created for.

5.2 Functional testing

Case	Test case	Expected result
T.4	Code functionality	The code for the weather rendering system compiles and runs without error.
T.5	User interface	The user is able to switch between the two modes, "real-life" and "sandbox". The user is also able to control the weather system over the user interface accordingly.
T.6	Performance	The shader code should run with reasonably good performance and should not show visual stutters or frame drops.

5.3 Visual testing

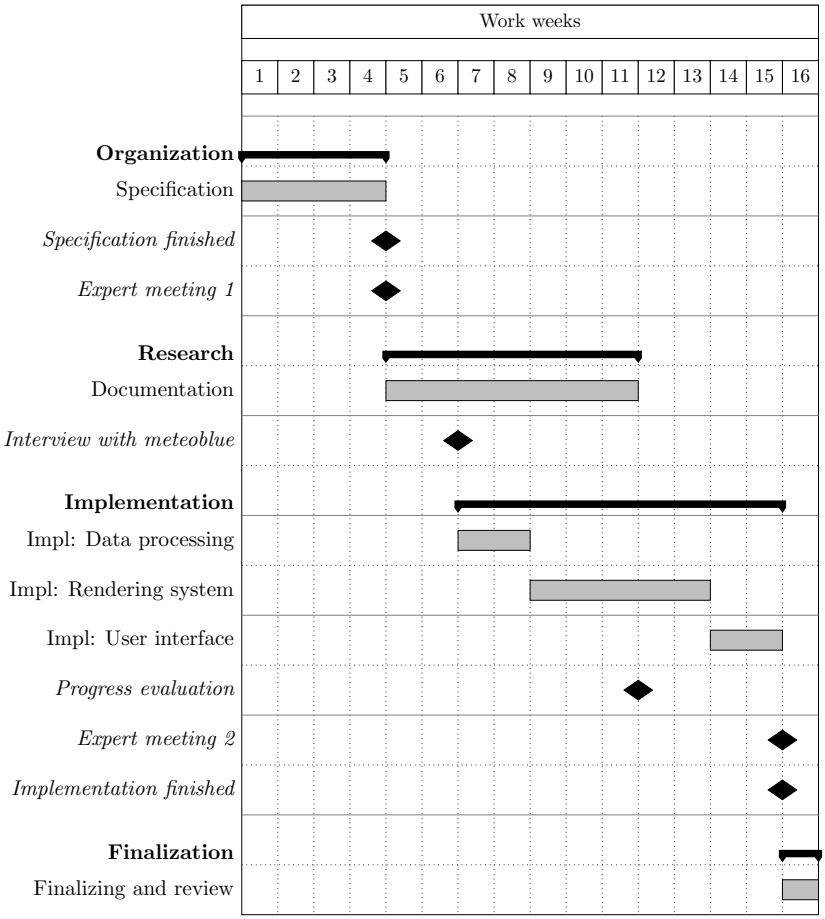
Case	Test case	Expected result
T.7	Real photographs	The visual output of the weather rendering system is to be compared with live weather photographs from *Roundshot* cameras. The rendered image should resemble the weather of that time, to a reasonable extent.
T.8	Similar products	The visual output of the weather rendering system is to be compared with the in-game footage of Microsoft's *Flight Simulator* game. The rendering system should achieve similar results, to a reasonable extent.

6 Project management

6.1 Schedule

The time frame of the semester spans over exactly 16 weeks. Being worth 12 ECTS points, this project assumes a maximum work load of 22.5 hours per week, resulting in a total of 360 hours.

Over the course of the term, the project will be split into four primary task groups, namely organization, research, implementation and finalization. Put into relation with the duration of the project, the estimated schedule looks like this:



6.1.1 Task Groups

For each task group, the following distribution of time and effort is estimated:

Task group	Predicted effort
Organization	10%
Research	40%
Implementation	45%
Finalizing	5%

The task groups are defined as follows:

- **Organization**
  The first task group focuses on creating and finishing the project specification. This also includes the first meeting with the thesis expert, Dr. Eric Dubuis.
- **Research**
  The research spans over the course of almost two months. It also continues being active during the first half of the implementation. This is necessary, as the topics will be further investigated when implementing them, which results in more research. Also, a technical interview with the company *meteoblue* will be scheduled and held during the first couple of weeks of the research task.
- **Implementation**
  After researching each relevant topic thoroughly, the implementation can begin. In this task, the weather system will be created in the Unity Engine. The implementation is split into three major categories, the first of which is the data aggregation and processing. Once that is done, the core system can be implemented. Finally, the UI will be implemented.

6.2 Project Organization

There are two kind of meetings during the project. For each, the protocol will be written in the project's journal. Should a physical meeting be impossible for some reason, an online meeting via Microsoft Teams will be held instead.

6.2.1 Weekly meetings

A meeting will be held on a weekly basis to discuss the progress of the thesis, encountered issues as well as planned work for the upcoming week.

Name	Role	Participation
Matthias Thomann	Author	Mandatory
Prof. Urs Künzler	Tutor and reviewer	Mandatory

6.2.2 Expert meetings

Additionally, both before the research task begins and after the implementation task has ended, a meeting with the external thesis expert will be held.

Name	Role	Participation
Matthias Thomann	Author	Mandatory
Dr. Eric Dubuis	Examination expert	Mandatory
Prof. Urs Künzler	Tutor and reviewer	Optional

6.3 Project Deliverables

The project results are the following items:

- **Documentation**
  The documentation includes this document as well as the thesis report.
  - Requirement specification
  - Thesis report
- **Implementation of the System**
  The Unity project, including all implemented shader code, will be managed and stored in the given GitLab repository [10]. This will also serve as a form of submission for grading.
- **Presentation**
  A public presentation will be held on the last Friday of the term, June 18, 2021.
- **Defense of the Thesis**
  The bachelor's thesis defense will be held after the term, on a day between June 21, 2021 and July 14, 2021. The exact date is yet to be arranged.

6.3.1 Submission Terms

The following items must be submitted.

Item	Description	Due Date
Specification	This document	March 19, 2021
Book entry	An advertising one-page description of the thesis	to be announced
Poster	An advertising poster of the thesis (A1 format)	June 7, 2021
Video clip	An advertising one-minute video clip of the thesis	June 17, 2021
Thesis	The thesis paper and all of the source code	June 17, 2021
Thesis print	The printed thesis including a CD with all source code	June 21, 2021

Glossary

Compute shader A shader which runs on the GPU but outside of the default render pipeline. 7

GPU Graphics processing unit. A piece of hardware designed to rapidly manipulate and alter memory, often intented for output to a display device. 16

HLSL High-level shading language. Developed by microsoft, this is a standard shader language for DirectX used in graphics programming. 8

Interpolation In mathematics, interpolation describes a method of estimating unknown values that fall between known values. 7

JSON Java-Script object notation. A light-weight data format that is stored as human-readable text. 4

LaTeX A high-quality document preparation system designed for the production of technical and scientific documentation. 16

Neural network A series of algorithms that can recognize and categorize certain patterns in a given set of data. 7

Noise A randomly generated pattern, referring to procedural pattern generation. 16

Noise generation Noise generation is used to generate textures of one or more dimension with seemingly random smooth transitions from black to white (zero to one). 6, 7, 10, 11

PNG Portable network graphic. A common format for lossless compressed image files. 4

Procedural Created solely with algorithms and independant of any prerequisites. 7, 16

Ray marching Ray marching is a type of method to approximate the surface distance of a volumetric object, where a ray is cast into the volume and stepped forward until the surface is reached. 6

Shader A piece of software which runs on the GPU, rendering geometrically defined objects to the screen. 6, 7, 8

SLPK ESRI scene layer package A custom, web-optimized format used for files related to ESRI. 4

UI User interface. The interface that allows the user to interact with the software. 5, 11, 14

Volumetric This describes a technique which takes a 3D volume of data and projects it to 2D. It is mostly used for transparent effects stored as a 3D image. 6, 11

References

[1] *Rendered image of volumetric clouds.* [Online]. Available: https://www.gosunoob.com/guides/change-live-weather-problems-in-flight-simulator-2020/.

[2] *Photographic reference of clouds.* [Online]. Available: https://bantiger.roundshot.com/.

[3] *Meteoblue: Weather data packages.* [Online]. Available: https://docs.meteoblue.com/en/apis/weather-data/introduction.

[4] *Meteoblue: Basic 1h packages.* [Online]. Available: https://docs.meteoblue.com/en/apis/weather-data/packages-api#basic.

[5] *Seitz: 360° roundshot camera system.* [Online]. Available: https://www.roundshot.com/.

[6] *Roundshot camera: Bantiger.* [Online]. Available: https://bantiger.roundshot.com/.

[7] *Roundshot camera: Gurtenpark.* [Online]. Available: https://gurtenpark.roundshot.com/.

[8] *Swisstopo: Elevation models.* [Online]. Available: https://www.swisstopo.admin.ch/en/geodata/height.html.

[9] *Swisstopo: Free geodata udpate.* [Online]. Available: https://shop.swisstopo.admin.ch/en/products/free_geodata.

[10] *Gitlab: Thesis repository.* [Online]. Available: https://gitlab.ti.bfh.ch/cpvr-students/cloud-shader.

B Previous Work: Procedural Cloud Shader

Procedural cloud shader

Project documentation

Field of Studies:	BSc in Computer Science
Specialization:	Computer perception and virtual reality
Author:	Matthias Thomann
Supervisor:	Prof. Urs Künzler
Date:	April 12, 2021
Version:	1.0

Berner Fachhochschule — Haute école spécialisée bernoise — Bern University of Applied Sciences

Abstract

Clouds contribute a great deal to the overall ambience in games and can be the cherry on top by filling the sky with life. To get as close as possible to real clouds, this project engages in researching and prototyping a procedural, volumetric cloud shader.
In order to achieve volumetric rendering, the document dives into the concept of ray marching, a group of methods used to render a 3D data set inside a container box to make it appear volumetric. Several variants of it are expanded on, like constant step, traditional, and sphere-traced ray marching. Additionally, to account for perception of depth, the volume can be shaded with the aid of surface normal estimation.
In the second part, 2D and 3D noise generation algorithms like Perlin's noise and the Voronoi algorithm are explained in detail. With fractal Brownian motion, the different layers of noise are then merged into one highly detailed noise texture.
At last, the goal of the project was to create prototypes in Unity displaying both volumetric rendering and noise algorithms, of which all were created successfully. Prepared with the combined knowledge of the research results and prototypes, a final shader was created, able to render a completely procedural and volumetric cloudscape.
For future work, the shader could be expanded into a fully-fledged weather simulation system with meteorologically accurate formation of clouds, rain, snow and much more.

Contents

1 General

1.1 Purpose

During this project, all gathered information and knowledge about the researched algorithms and techniques are written down. All prototypes and the final results are documented and compared with real photographs of clouds.

1.2 Audience

This document is written with the intent to further expand existing knowledge about the topic, hence it requires a fundamental knowledge about computer graphics and rendering.

1.3 Revision History

Version	Date	Name	Comment
0.1	March 21, 2020	Matthias Thomann	Initial draft
0.2	March 29, 2020	Matthias Thomann	Added first research results
0.3	April 01, 2020	Matthias Thomann	Added Unity prototype environment
0.4	April 03, 2020	Matthias Thomann	Added further research results
0.5	April 08, 2020	Matthias Thomann	Added further research results
0.6	April 13, 2020	Matthias Thomann	Added further research results
0.7	April 19, 2020	Matthias Thomann	Added research results about noise
0.8	April 26, 2020	Matthias Thomann	Added research results about noise
0.9	May 02, 2020	Matthias Thomann	Added Voronoi noise research
0.10	May 08, 2020	Matthias Thomann	Added FBM noise research
0.11	May 14, 2020	Matthias Thomann	Added prototype results
0.12	May 15, 2020	Matthias Thomann	Added prototype results
0.13	May 19, 2020	Matthias Thomann	Added prototype results
0.14	May 20, 2020	Matthias Thomann	Added prototype results
0.15	June 01, 2020	Matthias Thomann	Added realism checks
0.16	June 03, 2020	Matthias Thomann	Spelling revision
0.17	June 05, 2020	Matthias Thomann	Finalized document
1.0	June 12, 2020	Matthias Thomann	Finalized document after review

2 Natural Clouds

2.1 Formation

Clouds, as seen in nature, consist of a visible body of tiny water droplets and frozen crystals. In their natural occurrence, clouds are mostly generated from a nearby source of moisture, usually in the form of water vapor. This composition of particles creates the pleasant look of a white-grayish "fluffy" mass, floating in the sky.
Due to certain factors like altitude or water source, different types of cloudscapes can be formed. They vary in shape, convection, density and more. That makes different cloud types highly unique in terms of appearance.
For now, those factors are regarded as nature's randomness. However, an approximation of randomness will be covered in section 5.

2.2 Types of Clouds

Cloudscapes are given a genus and classified in multiple groups, mainly differing in altitude, meaning the distance from the earth's surface to the cloud formation. The following four cloud genera stand out due to their distinctiveness. A realistic simulation of a cloud system would consist of a combination of these types, which is why they are displayed here.

Figure 1: Photographic reference of stratus clouds [7].

Figure 2: Photographic reference of cirrus clouds [8].

Figure 3: Photographic reference of an altocumulus cloud formation [9].

Figure 4: Photographic reference of stratocumulus cloudscape [10].

3 Clouds in Games

Depicted in Figure 3 and Figure 4 of subsection 2.2 are clouds of the genus *cumulus*, which translated to English means *heap* or *pile*. Their remarkable cotton-like look makes them easy to recognize, which is also why they are often used in games as a reference for "normal" clouds. That is also the reason why the prototypes are mainly related to this genus.
In games, the formation as well as the natural composition are both irrelevant, as the clouds are essentially only used for cinematic ambience or as a mean to enhance the atmosphere. This leaves just the rendering technique and performance to worry about.

3.1 Skyboxes

A widespread solution for representing clouds in games is not rendering them at all, but instead using a set of polar sky dome images, also known as the skybox. This is a six-sided cube which is rendered around the whole game world. On each inward looking face of the cube, one of the sky dome images is displayed, creating a seamless sky around the inner side of the box.

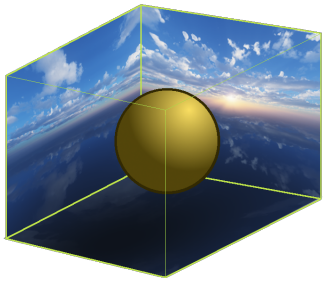


Figure 5: The skybox cube as it is used in games.

Figure 6: The polar sky dome images, folded out.

Besides rendering the sky, this of course allows clouds to be drawn right into the background. Also, in terms of performance, this is extremely cheap and efficient. On the other hand, it removes the ability for the clouds to move. They also have no volumetric body and no way of interaction with the game world.
This method does indeed give the scenery a more cloudy look, but what is missing is the "feel", or in other words the motion, interaction and lifelikeness of the clouds.

3.2 Billboards

Similar to the approach with the skybox, this technique also only uses 2D images of clouds. They are rendered individually and are always facing the camera. This is called *billboarding*. Now that each cloud is represented by its own game object, having a position in world space as well as a scale and many other properties, it is possible to animate the clouds. For example, by moving the game objects in a circle around the world, the clouds seemingly "pass by".

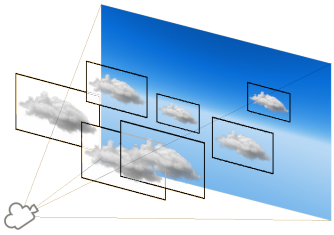


Figure 7: A collection of 2D cloud billboards facing the camera.

Figure 8: The rendered result of the image to the left.

Due to billboarding, the orientation is already given, making the overall rendering time and effort of this technique quite advantageous to others.
The major flaw of using billboards is of course that they are still 2D images, meaning they cannot really change appearance and therefore, do not evolve at all. Still, for many games, this technique suffices in the required diversity of background scenery and does not exceed the allowed performance share for such a task.

3.3 Mesh-based Objects

It is imaginable to simply use a polymesh shaped like a cloud and render that like any other game object. By adding a texture, this would make for some decent looking clouds.
However, the level of detail of such a polymesh is directly connected to the amount of vertices and faces that have to be processed every frame. As seen in Figure 9, there are hundreds of polygons required to merely represent the basic shape of a realistic cloud. If a similarly complex mesh is to be used for every cloud, a massive overhead is generated for objects that usually only contribute to the background of a game.

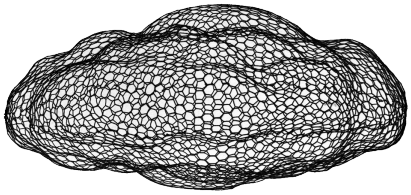


Figure 9: A polymesh in the shape of an altocumulus cloud [11].

Apart from the performance impact, this method offers a volumetric, possibly interactable object just like any other 3D model does. When massively decreasing the polygon count and therefore relinquishing the realistic look, mesh-based objects may be a viable solution for some low poly games. Otherwise, it is not reasonable to use this method.

3.4 Volumetric Clouds

Finally, clouds can be rendered via a technique called *volumetric rendering*. The images below show volumetric cloudscapes as seen in popular video games of major publishers. The method itself is explained in detail in section 4.

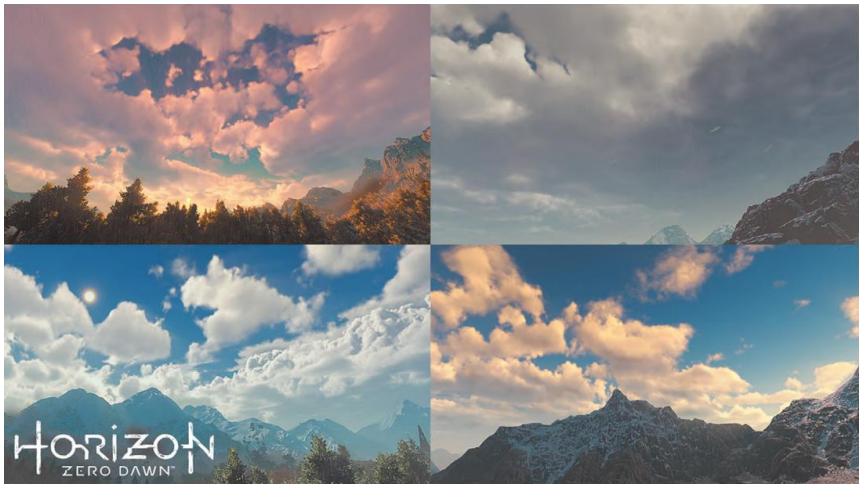


Figure 10: Several volumetric cloudscapes from the game *Horizon: Zero Dawn*, drawn in real time [12].

4 Volumetric Rendering

4.1 Definition

Volumetric rendering describes a technique for generating a visual representation of image data that is stored in a 3D volume. This especially comes to use for visual effects that are volumetric in nature, like fluids, clouds, fire, smoke, fog and dust, which are all extremely difficult or even impossible to model with geometric primitives.
In addition to rendering such effects, volumetric rendering has become essential to scientific applications like medical imaging, for which a typical 3D data volume is a set of 2D slice images acquired by a CT (computed tomography) or MRI (magnetic resonance imaging) scanner.

The data volume is also called a *scalar field* or *vector field*, which associates a scalar or vector value, called *voxel* (short for *volume element*), to every point in the defined space. For a scalar field, it can be imagined like a 3D grid, where each point holds a single number. This number could, for example, represent the density of a cloud at that very point. A vector field holds an n-tuple at each grid point.

4.2 Preliminary Notes

Most of the figures in the upcoming subsections depict only a single ray. However, this is only for explanatory purposes. The process has to be executed not once, but for each fragment processed by the fragment shader.

4.3 Constant Step Ray Marching

To actually render the volume data, a method called *ray marching* is used. With it, the surface distance of the volumetric data is approximated by creating a ray from the camera to the object for each fragment. The ray is then extended into the volume of the object and stepped forward until the surface is reached.

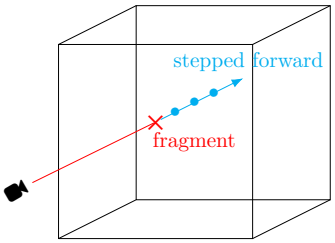


Figure 11: Ray marching concept visualized.

The ray-surface intersection is not directly calculated because it is not exactly defined for volumes like clouds, which is why the surface distance is approximated instead.

In ray marching, the algorithm only knows when it has reached the surface, or to be precise, when it is inside the actual object volume.
With this information, it is only possible to extend the ray in steps of a predefined length until the inside of the object is reached. With a constant step, the approximation of the surface distance is exactly as precise as the size of the constant step.
Once the ray is inside the actual volume, the functions returns the distance for this ray.

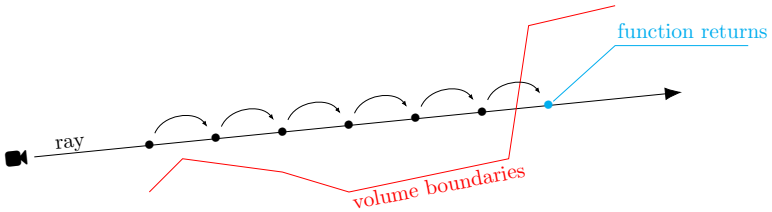


Figure 12: Traditional ray marching.

An implementation of this algorithm can be seen in Listing 1. Note that the volume to be rendered in this example is just a simple sphere. Also, the purple texts represent constants. Exact values for them are evaluated during prototyping.

```
// position:  the sampling point along the ray.
// direction: the ray's direction.
fixed4 raymarch(float3 position, float3 direction)
{
    for (int i = 0; i < MAX_STEPS; i++)
    {
        if (sphereHit(position))
            return fixed4(1,0,0,1);

        position += normalize(direction) * STEP_SIZE;
    }

    return fixed4(0,0,0,1);
}
```

Listing 1: Implementation of a ray march function with constant step.

In order to check if the ray is inside the volume, the function `sphereHit()` is used.

```
bool sphereHit(float3 position) {
    float4 sphere = float4(0, 1, 0, 1);
    return distance(sphere.xyz, position) < sphere.w;
}
```

Listing 2: Implementation of a volume distance function for a sphere.

4.4 Traditional Ray Marching

It is obvious to see that the constant step ray marching can only result in an accurate approximation of the surface distance if the step size is relatively small. This has a direct impact on performance and thus, is not a viable solution for the problem.
In traditional ray marching, an optimization has been developed for this problem. The algorithm does not blindly step forward, but instead tries to get as close to the real distance as possible. After the volume is reached, the step size is decreased and the ray steps out of the volume again. It then tries to approximate the surface distance by stepping in and out repeatedly in continuously smaller steps, thus converging towards the exact intersection. Once the step size falls below a certain threshold, the distance approximation is assumed to be precise enough and the value is returned for that ray march.

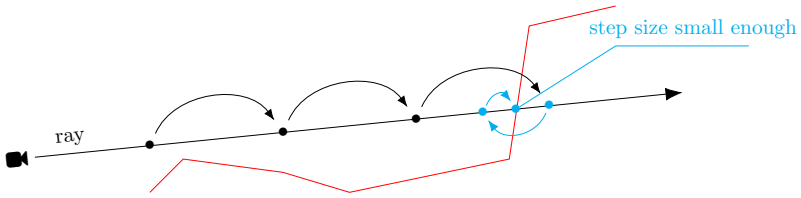


Figure 13: Traditional ray marching.

As visible, the traditional ray marching ends up with a more accurate result and the amount of steps per ray could be relatively lower, ultimately saving performance.
However, there is still an issue. The algorithm may jump in and out of the volume, even if it would already be precise enough, essentially taking unnecessary steps.

```
fixed4 raymarch(float3 position, float3 direction)
{
    float stepSize = STEP_SIZE;
    float dirMultiplier = 1;
    for (int i = 0; i < MAX_STEPS; i++)
    {
        if (stepSize < MINIMUM_STEP_SIZE)
            return fixed4(1,0,0,1);

        if (sphereHit(position)) {
            // reduce step size by half and invert marching direction.
            stepSize /= 2;
            dirMultiplier = -1;
        } else {
            dirMultiplier = 1;
        }

        position += normalize(direction) * stepSize * dirMultiplier;
    }

    return fixed4(0,0,0,1);
}
```

Listing 3: Implementation of a traditional ray march function with converging surface distance approximation.

4.5 Sphere Tracing

An even better approach to approximate the intersection of the ray and the volume is called *sphere tracing*. Instead of evaluating if the ray is inside the volume or not, an exact distance to the scene is measured. This distance is the minimum amount of space the algorithm can march along its ray without colliding with anything. For that, a function group called *signed distance functions* is used.

4.5.1 Signed Distance Functions

A signed distance function (SDF) returns the shortest distance from a given point in space to some surface. The sign of the returned value indicates whether that point is inside the surface or outside, hence the name.
For example, the signed distance function $f(p)$ for a point $p = (p_1, p_2, p_3)$ to the surface of a sphere $s = (s_1, s_2, s_3)$ with radius R looks like this:

$$f(p) = \sqrt{(s_1 - p_1)^2 + (s_2 - p_2)^2 + (s_3 - p_3)^2} - R$$

This translates into a simple code snippet, mostly identical to the function `sphereHit()` in Listing 2, except the distance is returned instead of a Boolean.

```
float sceneSDF(float3 position) {
    float4 sphere = float4(0, 0, 0, 1);
    return distance(sphere.xyz, position) - sphere.w;
}
```

Listing 4: Implementation of a signed distance function for a sphere.

With the sphere in the example being at the origin and having $R = 1$, a positive distance is returned for points outside the sphere and a negative distance if the point is inside the sphere.

```
float d1 = sceneSDF(float3(2, 0, 0));   // d1 = 1.0
float d2 = sceneSDF(float3(0, 0.5, 0)); // d2 = -0.5
float d3 = sceneSDF(float3(5, -5, 5));  // d3 = 7.66
```

4.5.2 Sphere Tracing with SDFs

If the distance to the scene can be calculated with a signed distance function, the algorithm becomes rather straight forward. The distance to the scene is evaluated at the start, then one can freely march along the ray for that amount of distance. Once arrived at the new point, the process is repeated until the SDF returns a small enough value.

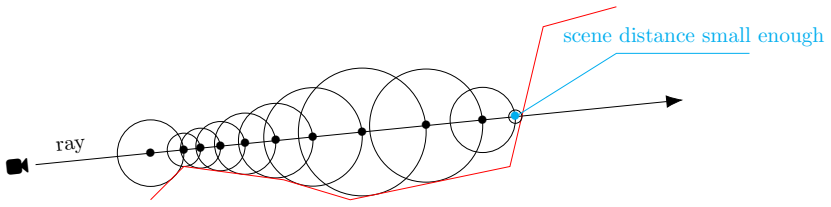


Figure 14: Ray marching with SDF-based sphere tracing.

As seen in Figure 14, the result is highly accurate. For the previous example with just one single sphere as a volume, the algorithm can be implemented like in Listing 5.

```
float raymarch (float3 position, float3 direction)
{
    float dOrigin = 0.0;
    for (int i = 0; i < MAX_STEPS; i++)
    {
        float dScene = sceneSDF(position + dOrigin * direction);
        if (dScene < SURFACE_DISTANCE || dScene > MAX_DISTANCE)
            break;

        dOrigin += dScene;
    }
    return dOrigin;
}
```

Listing 5: Implementation of ray marching with sphere tracing.

In order to save on performance, it is imperative to break the loop when `distanceScene` exceeds `MAX_DISTANCE`. This way, the distance evaluation for that ray can be stopped earlier than waiting for the loop to complete. Another example why this check is important can be seen in the next figure. The ray is terminated early, because it does not collide and never reaches the minimum surface distance.

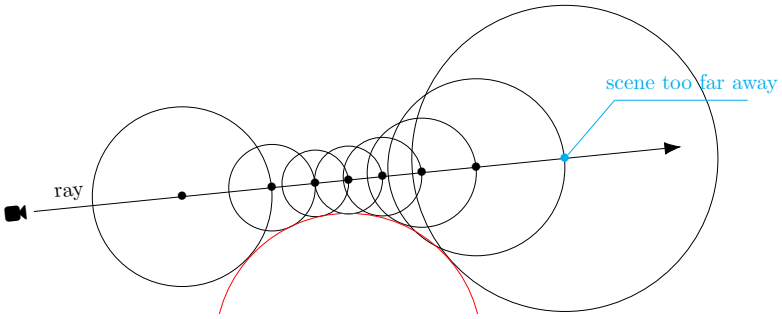


Figure 15: Ray marching with SDF-based sphere tracing, without collision.

4.6 Surface Normals and Lighting

As it is the case for many other lighting models, the surface normals are used to calculate lighting in volumetric rendering. If the object is defined with a polymesh, the surface normals are usually specified for each vertex. The normals for any given point on the surface can then be calculated by interpolating the adjacent vertex normals.
Since there is no polymesh in volumetric rendering, another solution has to be found for calculating the surface normals for a scene defined by signed distance functions. Because of that, it is not possible to explicitly calculate the normals and therefore, an approximation is used.

4.6.1 Surface Normal Estimation

To approximate the normal vectors in a 3D data volume, the *gradient* is used. The gradient represents the direction of greatest change of a scalar function. In Figure 16, the red arrows visualize the gradient. It is similar in 3D, where the gradient can be described as the path a ball would follow rolling downwards when dropped from the top of a corner.

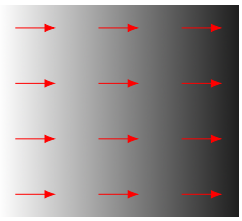


Figure 16: Gradient in a 2D scalar field.

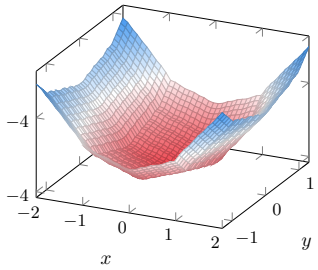


Figure 17: Gradient in a 3D scalar field.

Mathematically, the gradient of a function f at point $p = (x, y, z)$ defines the direction to move in from p to most rapidly increase the value of f. It is written as ∇f.

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

Instead of calculating the real derivative of the SDF, an approximation is used to estimate the normal vectors. As previously declared, the signed distance function returns zero for a point on the surface, greater than zero if the point is outside and less than zero if it is inside the volume. Therefore, the direction at the surface which will go from negative to positive most quickly will be orthogonal to the surface.

The estimation $\overrightarrow{n}$ is done by sampling some points around the point on the surface and take their difference, the result of which is the approximate surface normal.

$$\overrightarrow{n} = \begin{bmatrix} f(x+\epsilon, y, z) - f(x-\epsilon, y, z) \\ f(x, y+\epsilon, z) - f(x, y-\epsilon, z) \\ f(x, y, z+\epsilon) - f(x, y, z-\epsilon) \end{bmatrix}$$

The implementation of surface normal estimation looks like this:

```
float3 estimateNormal(float3 p) {
  return normalize(float3(
    sceneSDF(p + float3(EPSILON,0,0)) - sceneSDF(p - float3(EPSILON,0,0)),
    sceneSDF(p + float3(0,EPSILON,0)) - sceneSDF(p - float3(0,EPSILON,0)),
    sceneSDF(p + float3(0,0,EPSILON)) - sceneSDF(p - float3(0,0,EPSILON)),
  ));
}
```

Listing 6: Implementation of surface normal estimation.

Now that the normal vectors can be calculated for the volume, the object can be shaded. In this example, the Phong Illumination Model [13] is used.

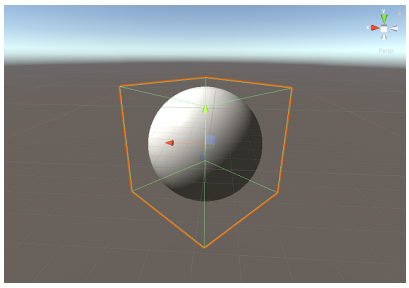


Figure 18: A 3D cube with a volumetric shader.

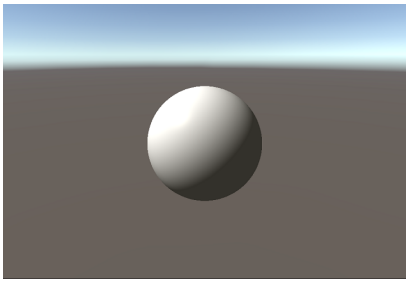


Figure 19: The shaded sphere rendered volumetrically.

4.7 Shadow Casting

In ray marching, rendering hard cast shadows proves to be rather easy. Naturally, the light ray comes from the sun, bounces off in the world and may eventually hit the eye of the observer. Since only a minute fraction of those rays actually reach the observer (the camera), a huge amount of rays would be calculated for nothing. Consequently, the rays are not traced from the light source to the camera but the other way around instead.
As defined in Listing 5, the `raymarch()` function moves along the given ray and returns the distance to the intersection point of ray and volume. Therefore, when a surface point has been determined, a second ray march can be started from the newly found point in the opposite direction of the primary light source's direction. If anything is hit on the way, the surface point lies in the shadow of the second hit object and should be darkened.

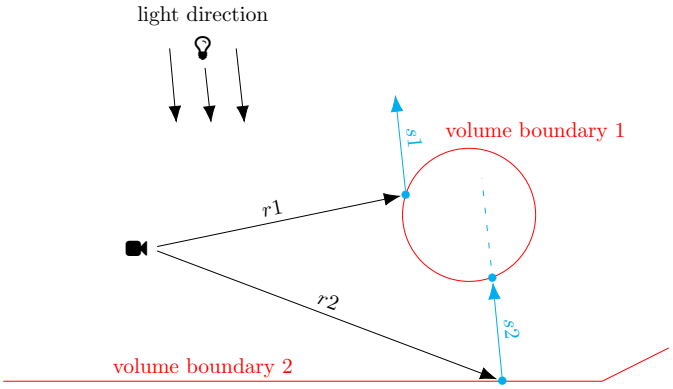


Figure 20: Shadow casting in ray marching.

As seen in the figure above, the ray $r1$ hits the volumetric sphere, then checks if anything is between the ray intersection and the negative light source direction. In this case, $s1$ does not collide with anything and the surface is shaded normally. For the other ray $r2$ however, the shadow ray march returns a distance $s2 > 0$ and $s2 <$ `MAX_DISTANCE`, meaning some object is in-between the hit point and the light source, casting a shadow.

```
// dMin: minimum distance required for shadow to be cast.
// dMax: maximum distance for shadow casting.
float hardshadow(float3 position, float3 direction, float dMin, float dMax)
{
    float dOrigin = dMin;
    for (int i = 0; i < MAX_STEPS; i++) {
        float dScene = sceneSDF(position + direction * dOrigin);
        if (dScene < SURFACE_DISTANCE)
            return 0.0;
        if (dScene > dMax)
            return 1.0;

        dOrigin += dScene;
    }
    return 1.0;
```

```
}
```

Listing 7: Implementation of hard shadow casting.

It is very clearly similar to SDF-based sphere tracing, except that only 0 or 1 is returned instead of the distance. The final color is then multiplied by this output. For 0, this results in a total black, hence the name *hard* shadows.

4.7.1 Soft Shadows

The method described in Figure 20 evaluates only if any given point is directly covered by any other object. It does not account for diffuse shadows with soft edges, called *penumbra* or simply *soft* shadows. But there is an easy and also cost-effective solution to that problem. Instead of strictly returning 0 when an object is covered by another, the shortest distance to the scene (qualified by some factor k) is returned.

```
float softshadow(float3 position, float3 direction, float dMin, float dMax,
    float k)
{
    float result = 1.0;
    float dOrigin = dMin;
    for (int i = 0; i < MAX_STEPS; i++) {
        float dScene = sceneSDF(position + direction * dOrigin);
        if (dScene < SURFACE_DISTANCE)
            return 0;
        if (dOrigin > dMax)
            return result;

        result = min(result, k * dScene / dOrigin);
        dOrigin += dScene;
    }
    return result;
}
```

Listing 8: Implementation of hard shadow casting.

Those are the resulting renders with a sphere and a flat box as the volumetric scene.

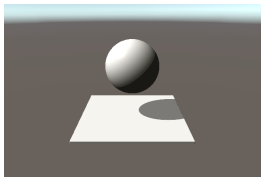


Figure 21: Hard shadows only.

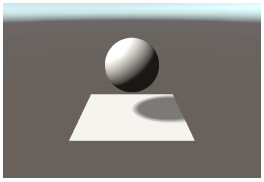


Figure 22: Soft shadows with $k = 7.0$.

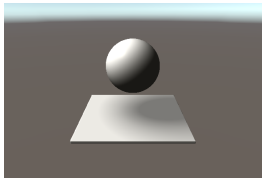


Figure 23: Soft shadows with $k = 1.2$.

4.8 Shape Blending

Another thing that comes free with ray marching is *shape blending*. It describes the concept of blending the signed distance functions of multiple shapes together with this simple method:

```
// d1: the first SDF.
// d2: the second SDF.
// k:  interpolation factor.
float blend(float d1, float3 d2, float k)
{
    return k * d1 + (1 - k) * d2;
}
```

Now two shapes can simply be blended like that:

```
float sceneSDF(float3 position)
{
    return blend(sphereSDF(position), boxSDF(position), 0.5);
}
```

The following image displays the two blended shapes. Due to the fact that the shadow calculation is based on the same SDFs, no additional changes have to be made in this regard.

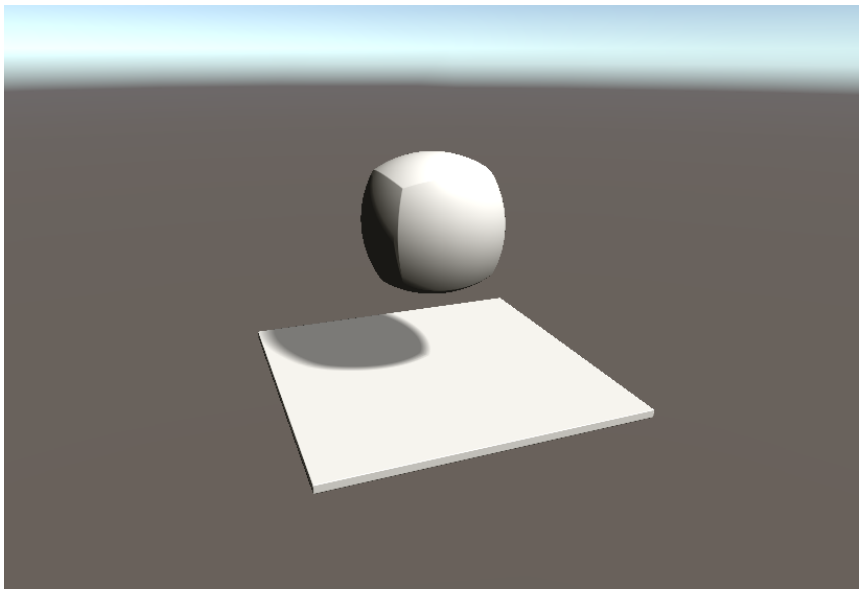


Figure 24: A blended sphere and box SDF with $k = 0.5$.

4.8.1 Constructive Solid Geometry

To create more interesting figures than a rounded box, constructive solid geometry (CSG) operators can be used. As seen in Figure 25, "holes" are cut into the geometry. This is done by taking the difference (or intersection) of the box and a cylinder that goes through the box. Like the `blend()` function takes in two signed distance function results, the following methods are also based on those values.

```
float intersection(float d1, float d2)
{
    return max(d1, d2);
}

float union(float d1, float d2)
{
    return min(d1, d2);
}

float difference(float d1, float d2)
{
    return max(d1, -d2);
}
```

Listing 9: Implementation of constructive sold geometry.

In this example, the intersection was done three times, for each axis once.

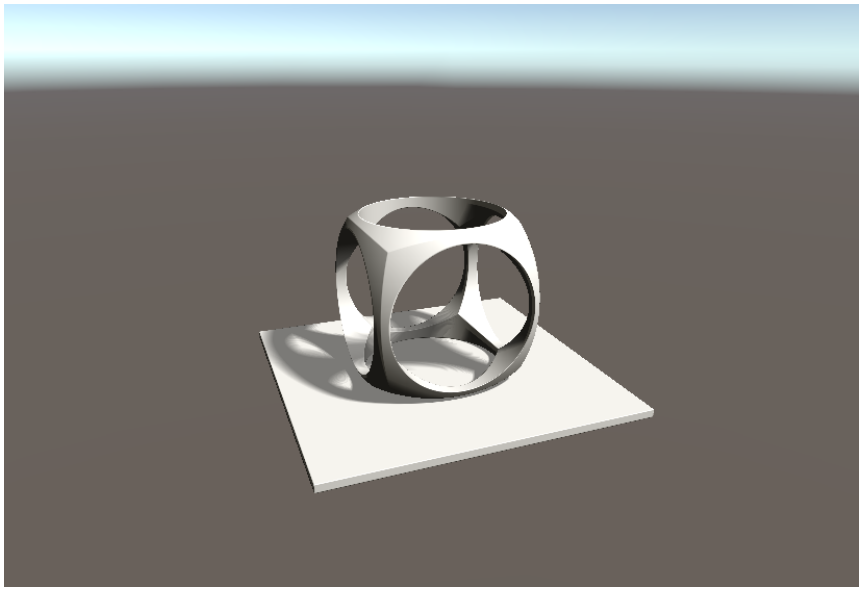


Figure 25: A blended sphere and box with cylinder intersection holes along each axis.

4.9 Ambient Occlusion

Shadow casting already looks quite realistic, but there is an important detail missing, called *ambient occlusion*. This method darkens areas around edges and crevices in the scene, making them look less exposed to the light and its environment. The algorithm for that is fairly uncomplicated and straightforward, given all the previously defined methods like `sceneSDF()` and `raymarch()` already exist.
When the `raymarch()` function returns a valid distance, a surface is hit. On that hit point p_1, the normal vector $\overrightarrow{n}$ is estimated. Now the distance to the nearest surface in the direction of $\overrightarrow{n}$ is evaluated. If on that ray a hit point p_2 is close, the color for the original hit point p_1 is darkened by some amount, depending on how far apart those points are.

```
1  float ambientOcclusion(float3 p, float3 direction) {
2      float ao = 0;
3      float dOrigin = 0;
4
5      for (int i = 1; i <= AO_ITERATIONS; i++) {
6          dOrigin = AO_STEP_SIZE * i;
7          ao += max(0, dOrigin - sceneSDF(p + direction * dOrigin)) / dOrigin;
8      }
9      return 1 - ao * AO_INTENSITY;
10 }
```

Listing 10: Implementation of ambient occlusion.

This comes close to the constant step ray marching algorithm, since it is marched along the ray in a predefined step size. On line 7, the scene SDF is substracted from the total distance and then divided by it. This just puts the scene distance in relation to the total distance. Also, `max()` is used because the SDF can return a negative number for points inside the surface, so in order to not brighten the scene at point `p` when this is the case, 0 is used instead.
With `AO_STEP_SIZE = 0.1`, `AO_ITERATIONS = 3` and `AO_INTENSITY = 0.2`, the following output is produced:

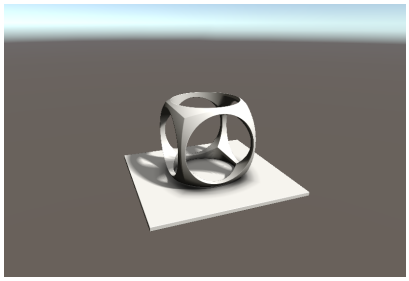


Figure 26: Ambient occlusion applied to the scene.

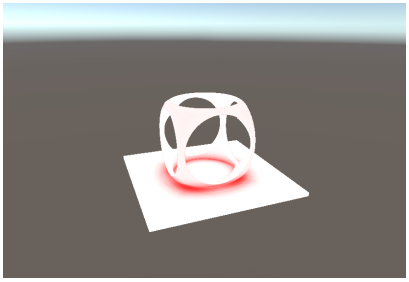


Figure 27: Only the ambient occlusion part drawn in red.

When comparing the previous Figure 25 with Figure 26, the darker ground around the object clearly improves the scene.

5 Noise Generation

Nature's chaotic behavior plays a big role in the diversity and appearance of cloudscapes. In shaders, an approach to simulate *randomness* is using so-called *noise generation*. In order to be able to implement random noise generation, several important topics need to be looked into. It is best to start with randomness in computer science and how it is handled inside a shader program.

5.1 Random Numbers

Unfortunately, there is no magic function which returns a purely random number inside the seemingly predictable and deterministic execution environment. So the question arises as to how such randomness can be generated.
For this, the function $rnd(x) = fract(sin(x))$ is inspected, where $fract(x) = x - floor(x)$.

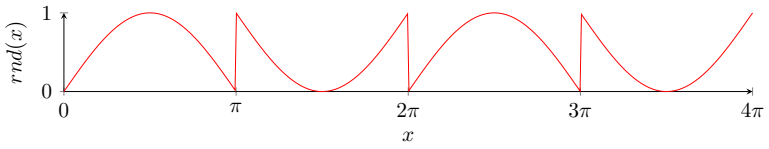


Figure 28: Random numbers with the fractional value of sine of x.

The sine values fluctuate between -1.0 and 1.0, but with $fract$, only the fractional part is evaluated, turning the negative values into positive ones. This effect can be used to get some pseudo-random values by "compressing" the function horizontally, or in other words by increasing the frequency of the sine wave.
The next figure displays the function $rnd(x) = fract(sin(x) * 10000)$.

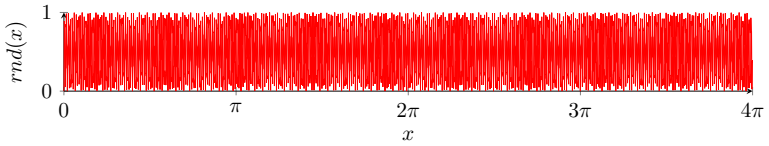


Figure 29: Random numbers with the fractional value of sine of x multiplied by 10000.

It is clearly visible that the function $rnd(x)$ became chaotic and returns practically random values. However, it is noteworthy that $rnd(x)$ is still a deterministic function, which means for example $rnd(1.0)$ is always going to return the same value.

5.2 2D and 3D Random

To generate a pseudo-random number from two and three values instead of one, the same function can be used, with some tweaks. Those two numbers come as a two-dimensional vector, which needs to be transformed into a single floating point number. According to Vivo [5], the dot product is particularly helpful in that case. It returns a single float value between 0.0 and 1.0 depending on the alignment of two vectors. They describe the following methods:

```
// co: two-dimensional position vector.
float random(float2 co) {
    float2 other = float2(12.9898, 78.233);
    return fract(sin(dot(co, other)) * 43758.5453123);
}
```

Listing 11: Implementation of 2D random number generation.

```
float random(float3 co) {
    float3 other = float3(12.9898, 78.233, 37.719);
    return fract(sin(dot(co, other)) * 43758.5453123);
}
```

Listing 12: Implementation of 3D random number generation.

When using the fragment coordinates as the vector `co` to call `random(co)` for every pixels, the resulting image shows a seemingly random assortment of pixels holding values from 0 to 1 (from black to white).

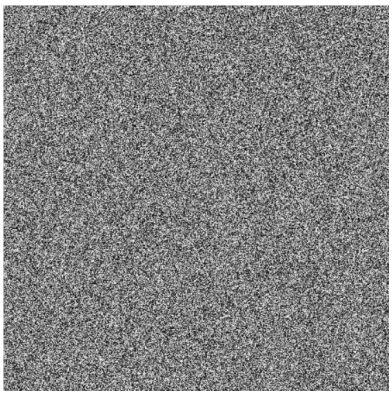


Figure 30: 2D random function visualized.

This method of procedural randomness still has one major flaw: It has no patterns. Contradictory to the word *random*, a certain pattern is required in order to generate *random* clouds. Luckily, there is more to random generation than just a high frequency sine wave.

5.3 Procedural Noise Patterns

Now that the concept of random numbers in the world of shaders is no longer a mystery, more advanced noise generation algorithms can be introduced. When using the word *noise* in this context, usually procedural pattern generation is meant.

5.3.1 Perlin Noise

One of the most commonly used procedural pattern generation algorithms is that of Ken Perlin. His algorithm works with the gradient, which was already introduced in subsubsection 4.6.1.
It consists of the following three steps:

1. grid definition
2. dot product calculation between random gradient and distance vectors
3. interpolation of those dot product values

Note that the following example refers to two-dimensional Perlin noise generation, but with some tweaks, is very much applicable for higher dimensional noise generation.
First, the 2D image space is split into a grid. For each vertex or corner point on this grid, a pseudo-random gradient vector is determined.

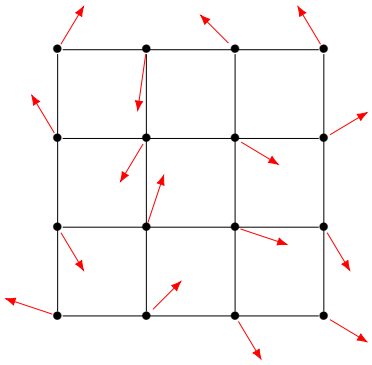


Figure 31: Perlin grid with pseudo-random gradient vectors.

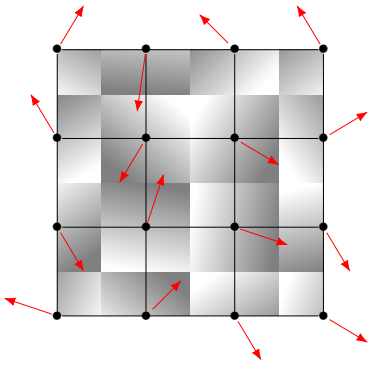


Figure 32: Perlin grid with visualized gradient vectors.

For the next step, it is easier to only inspect a single cell. Given the algorithm currently processes the highlighted pixel p in Figure 33, the next task is to determine the distance vectors from each adjacent corner point to the that pixel. Note that in $\mathbb{R}^2$, the amount of corners is four, while in $\mathbb{R}^3$, its eight.

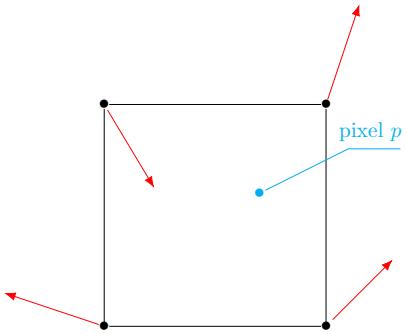


Figure 33: Perlin grid cell with gradient vectors.

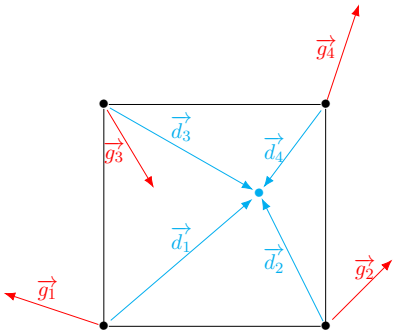


Figure 34: Perlin grid cell with distance vectors from each vertex to the pixel.

Then, the dot product is calculated for each distance vector and its gradient vector. This qualifies how similar those two vectors are, returning a positive number if they face the same direction and a negative one for the opposite. The dot product is 0 if the vectors are perpendicular.

$$
\begin{aligned}
s &= g_1 * d_1, \\
t &= g_2 * d_2, \\
u &= g_3 * d_3, \\
v &= g_4 * d_4.
\end{aligned}
$$

The values s, t, u, v represent the influences of the respective gradient on the final color of the pixel p. When visualizing those values as vectors with their length being the influence, it looks like this:

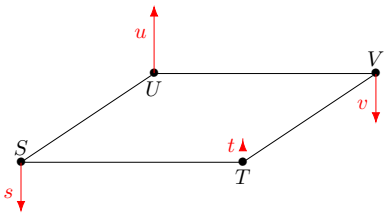


Figure 35: Perlin grid cell with visualized influences of gradient vectors.

It is clearly recognizable that the color of the pixel is influenced the most by u. Now those four numbers can be combined into one final number, the color value. For that, some sort of average calculation is used. For $\mathbb{R}^2$, the following ruleset applies:

1. find the average of the first pair of numbers
2. find the average of the second pair of numbers
3. average those two numbers together

To get an accurate mean value of those influences, rather than using the arithmetic average, a weighted average calculation is used. The weight for that is how close p is to the vertices. This means if p is close to a corner point, the influence of that vertex should be weighted heavier than the influences of all other corner points.
This is solved by linear interpolation.

$$d_x = (T_x - p_x)/(T_x) - (S_x),$$
$$d_y = (U_y - p_y)/(U_y) - (S_y).$$

$$w_1 = lerp(u, v, d_x),$$
$$w_2 = lerp(s, t, d_x),$$
$$w_{final} = lerp(w_1, w_2, d_y).$$

Both variables d_x and d_y represent the interpolation weight, being between 0 and 1. With w_1, the interpolation between s and t is done, depending on how far to the right the pixel is, related to its cell. This results in the first interpolation of the X-axis. Now w_2 is calculated, giving the second horizontal value in-between u and v. Finally, both w_1 and w_2 are linearly interpolated in relation to d_y, which gives the final average number.

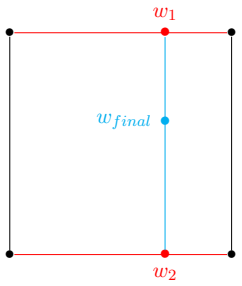


Figure 36: Perlin vertex weights in 2D space with four corners and three interpolations.

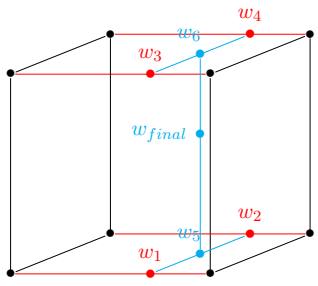


Figure 37: Perlin vertex weights in 3D space with eight corners and seven interpolations.

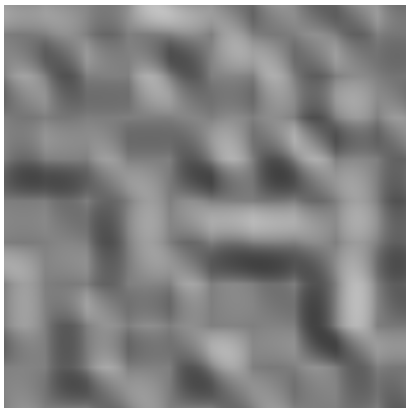


Figure 38: 2D Perlin noise texture with a 10x10 grid.

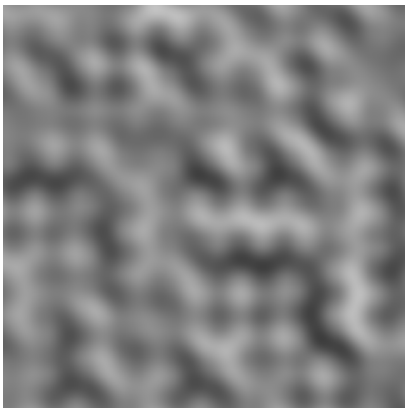


Figure 39: 2D Perlin noise texture with Perlin's fade function.

By default, the Perlin noise texture shows a significant amount of artifacts along the grid lines. This can be fixed by using Perlin's fade function [14] for d_x and d_y, which is defined by $f(t) = 6t^5 - 15t^4 + 10t^3$.

For 3D, Perlin describes that rather than calculating random gradient vectors, a simple set of 12 distinct vectors can be used, which still provides sufficient randomness but is faster [15]. For each grid corner, a hash function is used to generate an index (from 0 to 11), with which one of the gradient vectors is then chosen.

5.3.2 Voronoi Noise

While Perlin's noise algorithm is heavy on vector calculation and interpolation, other noise patterns are less complex to understand and construct, like the *Voronoi* noise, also known as *Worley* or *cellular* noise. The name derives from its similar structure to a Voronoi diagram, in which points, called *seeds*, are randomly scattered inside a defined space. After that, regions are created, consisting of all points closer to that seed than to any other.

As for the noise pattern, there are some alterations. To get a more even distribution, the noise algorithm starts by dividing the space into a grid, for which each cell is assigned a random point. From there, each fragment gets colored by how far it is to the seed in its cell.

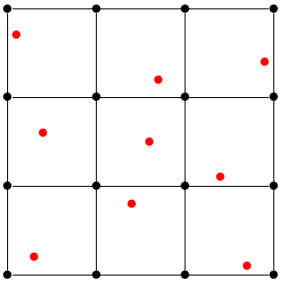


Figure 40: Voronoi grid with pseudo-randomly assigned seed points for each cell.

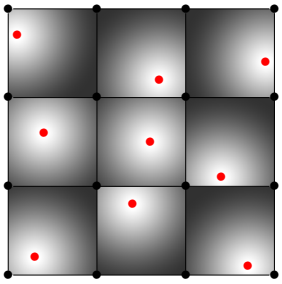


Figure 41: Voronoi grid with seed distances visualized.

As understandable, in Figure 41, hard contours are still visible along the grid lines. This can be improved by including the adjacent cells when finding the closest seed for any given fragment. This amounts to $3^n - 1$ neighboring cells, where n is the number of dimensions. This means for 2D space its eight cells, while in 3D its 26.

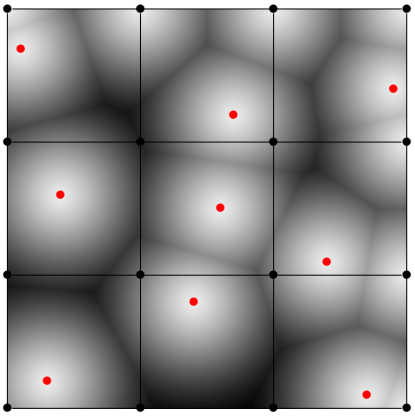


Figure 42: Complete 2D Voronoi noise pattern.

An implementation of this relatively simple algorithm could look like the following listing. `randomSeed()` is used like the previously introduced function `random()`, except that it returns a two-dimensional vector instead of a scalar. With that, a deterministically random point can be generated for any given cell.

```
float2 randomSeed(float2 co) {
    return float2(
        fract(sin(dot(co, float2(12.9898, 78.233))) * 43758.5453123),
        fract(sin(dot(co, float2(39.3461, 11.135))) * 14375.8545359));
}

float voronoi(float2 p) {
    float2 pCell = floor(p);
    float dMin = 999;

    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            float2 cell = pCell + float2(x, y);
            float2 seed = cell + randomSeed(cell);
            float d = distance(seed, p);
            if (d < dMin) {
                dMin = d;
            }
        }
    }

    return dMin;
}
```

Listing 13: Implementation of 2D Voronoi noise algorithm.

Since the Voronoi noise algorithm creates a cellular pattern, it is well suitable for simulating natural distribution of cloud heaps, as they are in some way also formed "in cells".

5.3.3 Fractal Brownian Motion

In the world of shaders, the term *fractal Brownian motion* (fBm) is often described as adding different levels of noise together, thus creating a self-similar pattern across different scales [16]. This simplified description meets the required level of detail for this section, a complete explanation and derivation of the fractal Brownian motion is beyond the scope of this paper.

In shaders, fBms are also called *fractal noise*. They are usually implemented by adding different iterations of noise (called *octaves*), while successively incrementing the frequencies in regular steps (*lacunarity*) and decreasing the amplitude (*gain*) of the noise. This results in a more detailed noise, meaning a finer granularity of the pattern in the noise.

```
#define LACUNARITY 2.0
#define GAIN 0.5
#define OCTAVES 1

float fbm(float2 p) {
    float frequency = 1.0;
    float amplitude = 0.5;

    float total = 0;
    float maxValue = 0;
    for(int i = 0; i < OCTAVES; i++) {
        float current = noise(p * frequency) * amplitude;
        total += current;
        maxValue += amplitude;

        amplitude *= GAIN;
        frequency *= LACUNARITY;
    }

    return total/maxValue;
}
```

Listing 14: Implementation of fractal Brownian motion function.

Interestingly, the only things that change for 3D is `float2 p` becomes a `float3 p` and the `noise()` function must accept a three-dimensional vector instead. That's all.
Here are some example images of the fractal Brownian motion with different octaves. For the noise function, a Voronoi noise algorithm was used.

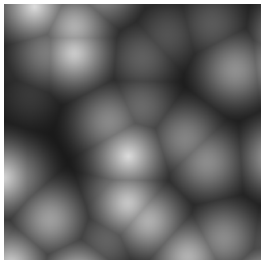


Figure 43: One octave of a 2D Voronoi noise.

Figure 44: Two octaves of a 2D Voronoi noise.

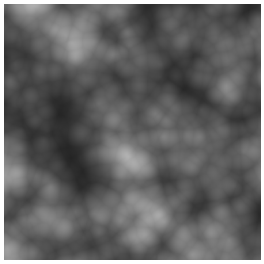


Figure 45: Three octaves of a 2D Voronoi noise.

It is understandable that with every additional octave, the algorithm has to evaluate the noise at all points again, making it worth considering the impact on performance fractal noise has. However, the final renders look convincingly "cloudy".

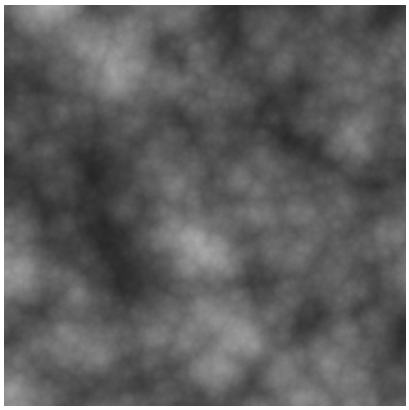


Figure 46: Ten octaves of a 2D Voronoi noise.

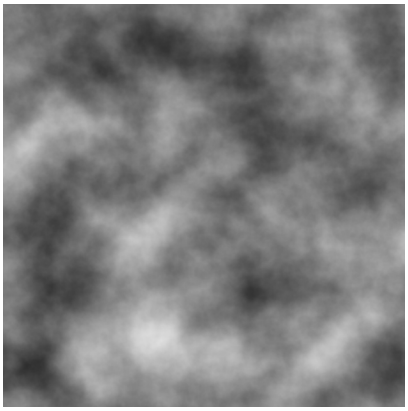


Figure 47: Ten octaves of a 2D Perlin noise.

6 Prototypes and Results

6.1 Preliminary Notes

6.1.1 Completed Prototypes

While researching the topic and experimenting with some dummy shaders, it came clear that "volumetric rendering" and "ray marching" are interchangeable in this matter. Therefore, only two kinds of prototypes have been developed. This change is explained in detail in subsection 8.2.

6.1.2 Dimensions

All of the following documented procedures and algorithms were prototyped and implemented in 3D, but for the matter of explanation, it is described and visualized in 2D.

6.1.3 Unity Variables

The following sections will list code snippets, in which all variables prefixed with an underscore are shader variables exposed to the Unity Editor. This way, they can be changed externally while running the shader code, allowing for convenient debugging. They are from here on referred to as *parameters*.

6.2 First Draft

The first drafts of prototypes created during this project are all related to volumetric rendering. Instead of using a signed distance function and evaluating the distance to a 3D data volume, a noise function was used to simulate the *mass* of the cloud. The primary issue was to get the cube transparent where the noise function would return a number close to 0.0 and to color it where the number would be close to 1.0. The approach for solving this issue is done by sampling the cloud's density.

6.2.1 Density Sampling

Like in volumetric rendering, for each pixel fragment, a ray is cast from the fragment into the cube and extended along the view direction for that fragment. Usually, the algorithm can stop for a given ray if the signed distance function returns a small enough distance, meaning the ray has hit a surface of the volume. However, it is different in the case with clouds, where the volume is *translucent* at most points.
To account for that, the ray does not stop until the end of the container box is reached. It samples the density N times along its path and returns the sum of those samples, giving an approximate qualifier for how dark this fragment should be.

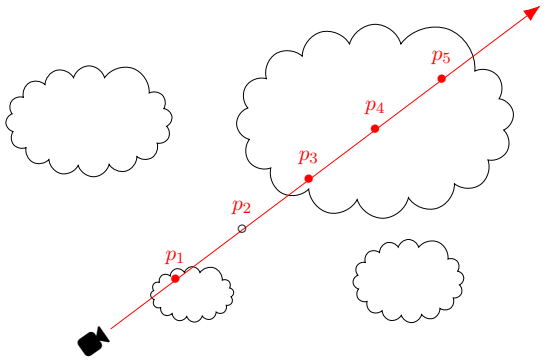


Figure 48: Density sampler ray with $N = 5$.

Understandably, the bigger clouds in Figure 48 represent higher return values of the noise function, meaning denser areas. For the displayed ray, the values for points p_1, p_3, p_4 and p_5 are accumulated and used as a qualifier to color the fragment. In this case, a rather dark tone would be used.
It is notable that N has an exponential impact on the performance, so it should be chosen carefully.

While marching along the ray, the step size is not constant but instead calculated:
$d_{step} = \frac{d_{box}}{N}$, where d_{box} is equal to the total distance the ray travels while inside the box. To determine d_{box}, an axis-aligned bounding box (AABB) intersection test [17] has to be done with the container box and the ray.

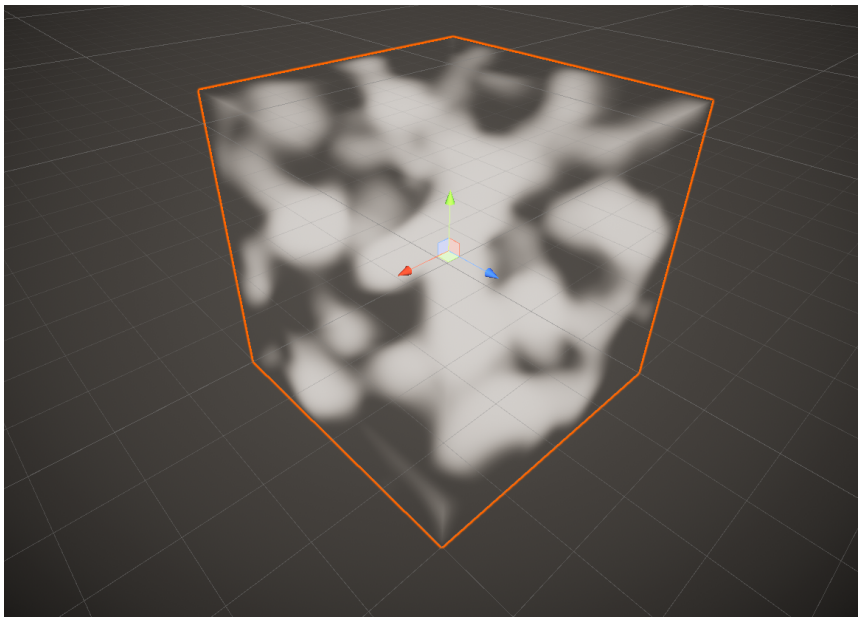


Figure 49: Prototype: Rendered image of sampled density based on 3D Perlin noise.

With this first try, a Perlin noise function was sampled. The returned value had to be normalized in a range of $[0, 1]$ in order for it to be used as alpha value of the color.

6.2.2 Normalizing Density

This is where the exponential function $exp(x) = e^{-x}$ comes in, which (when clamped between 0 to 1) converts very low values to 1.0 and higher values will converge towards 0.0.

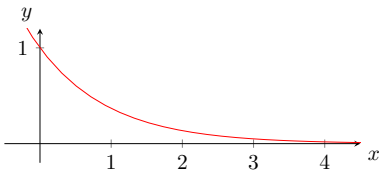


Figure 50: Exponential function $exp(x) = e^{-x}$.

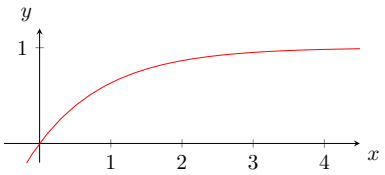


Figure 51: Inverted exponential function $exp'(x) = 1 - e^{-x}$.

When inverting $exp(x)$, the function $exp'(x)$ returns a value that can be directly used for the transparency of the cloud. The denser it gets, the more opaque it will be.

6.3 Improving Noise

After further experimenting with the noise sampling function, the idea arose to combine Perlin and Voronoi noise, which hopefully would create a more distinguished, random pattern. The final sampling function simply multiplies both noise values at a given point `position`, masking them with each other.

```
float sampleDensity(float3 position) {
  float3 vpos = position * _VoronoiScale + _VoronoiOffset;
  float3 ppos = position * _PerlinScale + _PerlinOffset;
  float vd = fbmVoronoi(vpos, _VoronoiOctaves, _VoronoiPersistence));
  float pd = fbmPerlin(ppos, _PerlinOctaves, _PerlinPersistence));

  vd = max(0, vd - _VoronoiDensityThreshold) * _VoronoiDensityMultiplier;
  pd = max(0, pd - _PerlinDensityThreshold) * _PerlinDensityMultiplier;

  // fixed boost for density by factor 2
  float density = vd * pd * 2.0;
  return density;
}
```

Listing 15: Implementation of a density sampling function.

By adjusting some of the parameters and increasing the octaves of both noises, a more patchy and cloudy look can be achieved at the cost of performance.

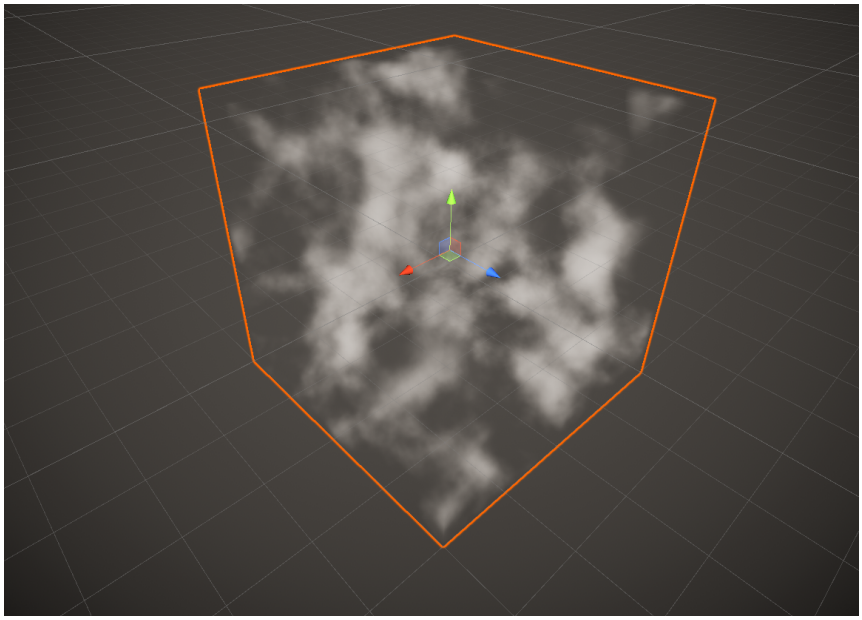


Figure 52: Prototype: Rendered image of sampled density based on mixed noises.

6.4 Light Transmittance and Light Scattering

One of the more prominent lighting features of clouds is its translucency. This phenomenon displays how light bounces and scatters inside the matter, then exits at a different point. This is also called *subsurface scattering* (SSS). It results in illuminated areas where the clouds are thinner, giving it that milky look and "glow" on the outer edge. In nature, subsurface scattering is a very complex and computationally demanding process. For computer graphics however, it is often either simplified or substituted with some other algorithm that produces a similar outcome at lower performance cost.

6.4.1 Sunlight Forward Scattering

When approaching the implementation of subsurface scattering and directional lighting, it seemed most reasonable to start with the sun being visible behind the clouds, or at least shining through them. This implies finding a way to illuminate clouds that cover the sun. In the context of this project, it is called *sunlight transmittance* or *sunlight forward scattering*, since it is not a variant of SSS but rather an approximation.
After some consideration and brainstorming, the following method was chosen to solve the issue:

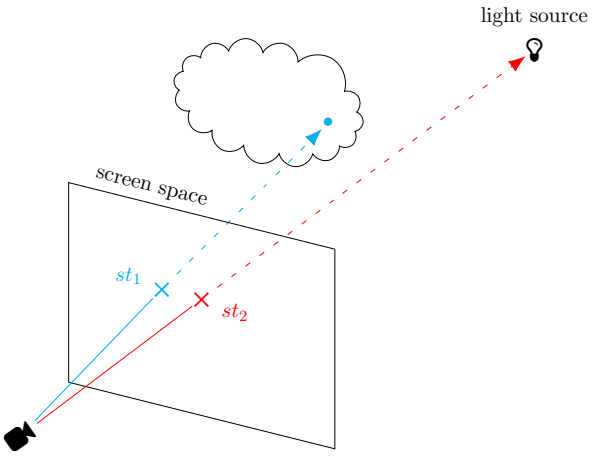


Figure 53: Sunlight transmittance sampling.

When ray casting, both the fragment's and the light source's screen-space position is calculated. Those are two-dimensional coordinates relative to the screen that the camera renders to. Now if the distance $d = \left\|\overrightarrow{st_1 st_2}\right\| < t$, with t being some threshold, a portion of the sun's color is added to the fragment's color, relative to how small d is.
It is noteworthy that when calculating the screen-space position, the depth value gets lost. Therefore, theoretically, the clouds would be illuminated when $d < t$ even if the sun is in front of the clouds. Given this is almost never the case in games and weather simulations, that particular issue is neglected.
Also, instead of evaluating the distance d, the intermediate angle of both vectors could also be used to avoid calculating the screen-space position, giving $d = cos^{-1}\left(\overrightarrow{v_{cloud}} * \overrightarrow{v_{light}}\right)$.

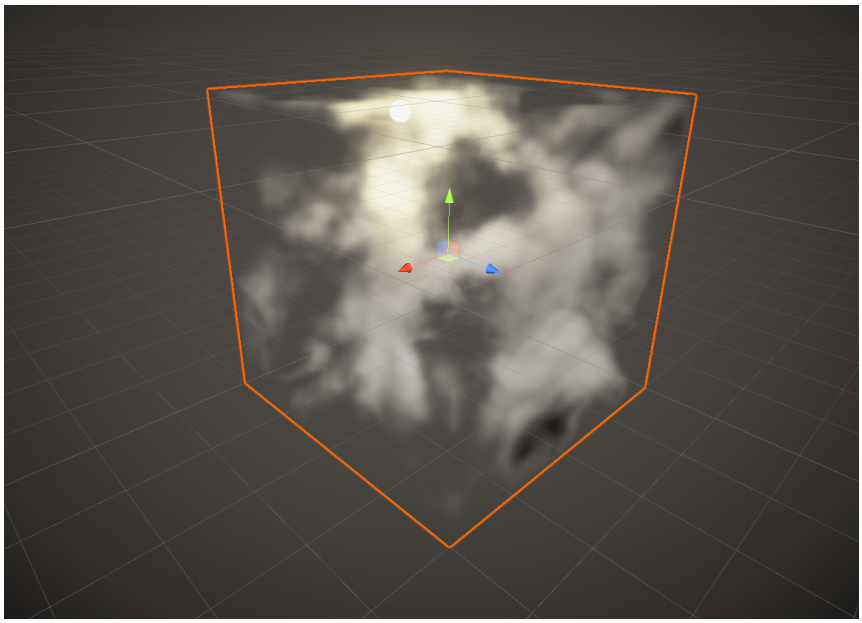


Figure 54: Prototype: Rendered image of sunlight transmittance.

Behind the cube in Figure 54 is a sphere object placed, representing the sun. The sunlight is indeed shining through the clouds, but there are still some minor flaws with the implementation. For example, some clouds are completely illuminated, making them too bright where the cloud would be too dense for the light to pass through.

The following code snippet shows the implementation of the sunlight transmittance mechanism. The `density` variable is the one evaluated in Listing 15.

```
1 float cloudDensity = exp(-density);
2
3 float projectedSunDistance = length(
4   worldToScreenPos(_SunPosition) - worldToScreenPos(worldPosition));
5
6 float sunTransmittance = 1 - pow(
7   smoothstep(0.01, _SunLightScattering,projectedSunDist), _SunLightStrength);
8
9 fixed3 sunColor = sunTransmittance * _LightColor0.xyz * cloudDensity;
```

Listing 16: Implementation of a sunlight transmittance mechanism.

Like in other prototype code listings, there are some parameters to play with. The sunlight strength and the its range in screen-space can both be adjusted, for example.
The idea of multiplying by `cloudDensity` on line 9 was to fix the previously described flaw of clouds being too bright.

6.4.2 Directional light

Another challenging part during prototyping was directional light on surfaces facing the sun. Usually in ray marching, a surface normal estimation is done using the gradient. This works well if there is only one point of interest (like a ray-surface intersection point), but as already mentioned before, the ray does not stop sampling points until it reaches the end of the container box.
So instead of calculating normals for each sample point, another ray is cast from the sample point towards the sun. Along its path, the density is sampled again L times in constant steps. With the lack of an official term, this process is called *lightmarching* in this project.

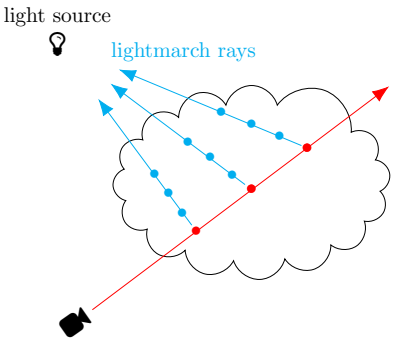


Figure 55: Directional lightmarching samples (part 1).

It is clearly visible that in Figure 55, a lot of density samples return a high value, resulting in a dark fragment color for this ray. To simplify, there is a lot of cloud mass in front of that sample point, so the fragment will not receive a lot of sunlight color.
On the other hand, in Figure 56, only very few samples are even inside a cloud, resulting in an overall low value. This leads to a higher influence of the sun's color for that fragment, meaning the samples are more exposed to the sun.

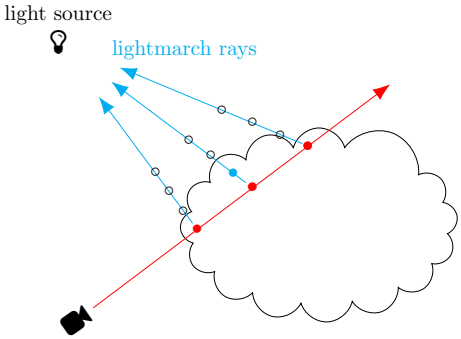


Figure 56: Directional lightmarching samples (part 2).

The implementation for lightmarching is rather straight-forward, given the concept of ray marching is already known.

```
float lightmarch(float3 position, float3 direction) {
    float3 p = position;

    float lightTransmittance = 0;
    for (int j = 0; j < _MaxLightSteps; j++)
    {
        p += direction * _LightStepSize;
        lightTransmittance += sampleDensity(p);
    }

    return lightTransmittance;
}
```

Listing 17: Implementation of lightmarching.

The method is called during ray marching and the original function is modified like below:

```
float2 raymarch(float3 position, float3 direction)
{
    float3 sunDirection = normalize(_SunPosition - position);
    float lightStepSize = insideBoxDist / _MaxLightSamples;
    float lightTransmittance = 0;

    [...ray marching...]

    for (int j = 0; j < _MaxLightSamples; j++)
    {
        position += direction * lightStepSize;
        lightTransmittance += lightmarch(position, sunDirection);
    }

    return float2(density, lightTransmittance);
}
```

Listing 18: Implementation of raymarching with lightmarching.

Now, two values are returned instead of just one. Both are later normalized with either $exp(x)$ or $exp'(x)$.

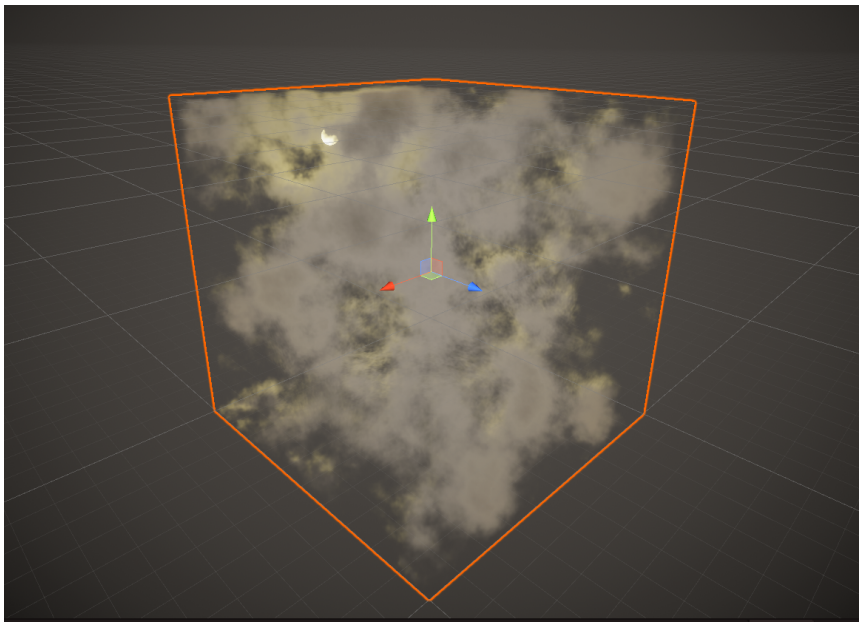


Figure 57: Prototype: Rendered image of directional sunlight implemented with light-marching.

6.5 Final Prototype

All put together and after quite some effort and experimenting, the rendered scene looks quite convincing.
Free assets from the Unity Asset Store were used for trees[1] and rocks[2].

Figure 58: Prototype: Rendered image of the final prototype (afternoon scene).

[1] https://assetstore.unity.com/packages/3d/vegetation/speedtree/free-speedtrees-package-29170
[2] https://assetstore.unity.com/packages/3d/environments/landscapes/photoscanned-moutainsrocks-pbr-130876

To demonstrate the capability of the shader in its prototype state, here are some variations of it. The are no code changes in-between the rendered images, the only things that changed are the shader's parameters and Unity Editor lighting settings and colors.

Figure 59: Prototype: Rendered day scene.

Figure 60: Prototype: Rendered puffy sky scene.

Figure 61: Prototype: Rendered night scene.

Figure 62: Prototype: Rendered sunset scene.

Figure 63: Prototype: Rendered clear sky scene.

Figure 64: Prototype: Rendered stormy scene.

6.5.1 Additional Masking

Technically, by multiplying both noise function values in `sampleDensity()`, they already mask each other. In certain types of cloud formations, an additional masking needs to be applied in order to create a cloudscape that does not expand over the whole container box. This is done by feeding a mask texture into the shader, for which the cube's UV coordinates are used to sample the grey value of the texture at that position.

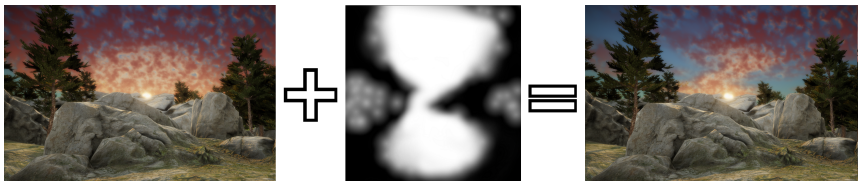


Figure 65: Prototype: Masking with UV coordinates of the container box.

6.6 Realism Check

While the prototypes in Figure 59 to Figure 64 do look realistically to a certain degree, it is still essential to have some sort of measurable factor with which the rendered images can be compared to real clouds. Some factor that ultimately shows how *real* the rendered clouds actually are, apart from the human eye interpretation.
Before expanding on how to measure the realism of the cloud shader's output, it seems important to objectively identify the capabilities of it first. As originally stated in section 3, the desired look of the clouds was that of the genus *cumulus*.
In the following subsection, each rendered image is compared to a real-life photograph, putting them into a objectively comparable state to reality.

6.6.1 Real Life Comparison

In all of the following comparison images, the left images are photographic references and the right side images are rendered in Unity Engine.
As for the cloud genus, Figure 59 and Figure 60 both resemble cirrocumulus and altocumulus clouds. The cirrocumulus clouds are similar to altocumulus clouds, but they form at higher altitude and are significantly smaller, yet equally puffy. Figure 61 and Figure 62 also show the distinctive cotton-like pattern of the altocumulus genus.

By adjusting the scale of the shader, the clouds can be made smaller. In the following comparison, the directional lightmarching has been turned down to a minimum so the light would look more pale.

Figure 66: Comparison: photographic reference [18] versus the rendered image.

The night-time comparison displays how the different color of the sunlight forward scattering can impact the scene.

Figure 67: Comparison: photographic reference [19] versus the rendered image (at night).

In Figure 62, it is very clearly recognizable that the parameters of the shader can heavily influence the lighting and illumination of the clouds. Unfortunately, the difference of details and density is fairly distinguishable from reality and the desired appearance is not fully achieved.

Figure 68: Comparison: photographic reference [20] versus the rendered image (at sunset).

Finally, Figure 64 represents clouds of the type *nimbostratus*, which form in low altitude and are dense and dark, often rainy.

Figure 69: Comparison: photographic reference [21] versus the rendered image (stormy).

With the sunlight transmittance being a bright yellow in the rendered image, an attempt was made to make the sun shine through the distant rainy clouds, like in the photographic reference. Since the cloud shader does not end before the horizon, the resulting effect is rather imperceptible.

6.6.2 Convolutional Neural Network

Given there is a convolutional neural network (CNN) that is able to classify images of the sky, the weather or clouds into descriptive labels or even genera of cloud formations, then one could just seed those rendered images into the CNN and verify whether the results are truthfully showing "real" clouds. Of course, this is heavily dependent of how well the CNN was trained.

6.6.3 Generative Adversarial Network

A similar approach to the convolutional neural network is a generative adverserial network (GAN) setup. It describes two neural networks, which compete with each other in a cat-and-mouse game: The *generative* network tries to imitate the training set by generating artificial photographs with many realistic characteristics, while the *discriminative* network tries to tell whether the generated images are fake or not.
With this method, the rendered cloud images could be passed through the discriminative neural network to see if at least the network thinks the images are of real clouds.

6.6.4 Histogram Comparison

The *histogram* is a graphical representation of data like brightness or color distribution of a given photograph. When extracting the color histograms of the real photograph and the one of the rendered image, they could be compared and rated how different in color they are.

6.6.5 Professional Meteorological Assessment

Another viable solution is to let a professional meteorologist inspect and rate the rendered images and judge the realism of the depicted scenarios, which should reveal if the rendered clouds could actually form and exist in reality.

6.7 Performance

With the current implementations of ray marching and lightmarching, the performance of the shader is heavily dependent on the number of samples N and L taken along the rays. With a container box the size of 400x200 pixels in screen-space (which is a relatively small box of clouds), the GPU load is already enormously large. Here, the number of noise samplings N_{total} is calculated as follows. The values for N and L are both set to a minimum of 25, since it seemed to achieve the best looking results during prototyping.

$$w = 400, h = 200$$
$$N = 25, L = 25$$

$$N_{total} = w * h * N * L = 5 * 10^7$$

So for a single frame, the number of times the noise function is called for this example is just about fifty million times. This inconceivably large number makes the current implementation desperately needy of optimization.

6.7.1 Optimization Attempts

6.7.1.1 Compute Shader

The fact that the noise function needs to be sampled so often may not even be the issue, but rather that the noise texture needs to be calculated again each time. Therefore, the idea arose to precalculate the 3D noise volume and feed it to the shader at runtime in the form of a 3D texture cube, hoping that when sampling the same position twice, the noise would only be calculated once.
For this, a compute shader was created with the attempt to generate the needed texture. Unfortunately, all experiments with compute shaders were unsuccessful, as there is only very little documentation about 3D texture-generating compute shaders for Unity. This attempt was therefore abandoned.

6.7.1.2 Early Exits

For some functions and loops, early exit conditions can speed up the shader. As an example in Listing 18, which describes ray marching combined with lightmarching, the light does not need to be calculated when there is no cloud. The following little extension should fix that issue.

```
1 if (density <= 0) {
2     return float2(0, 0);
3 }
```

Instead of checking if `density` is smaller or equal to zero, a very small threshold could be used as well. It is noteworthy that a number larger than zero can create edges on thin clouds, as the abrupt reduction in density may be visible.

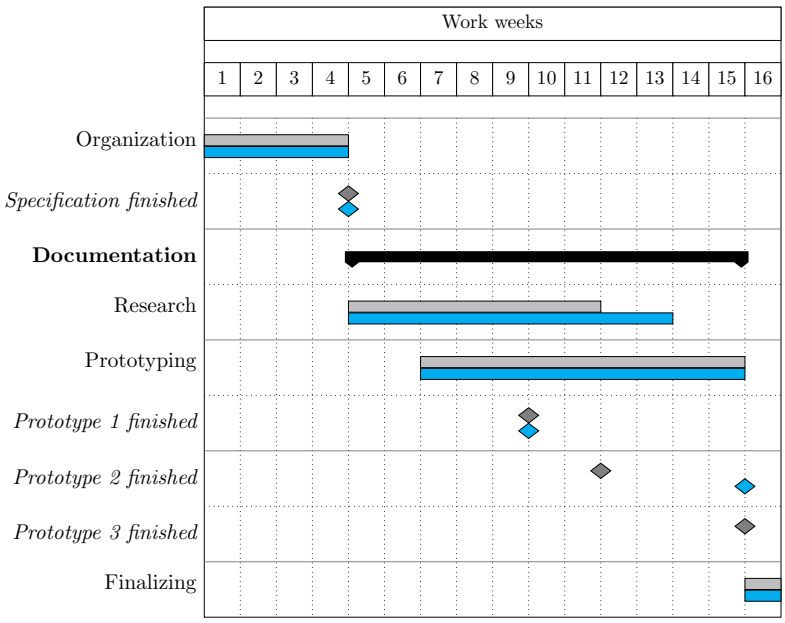
7 Conclusion and Critical Discussion

Viewed from a critical perspective, two out of three prototypes have been developed during the project, of which one was dropped because it was too similar to another. Additionally, a complete cloud shader has been programmed, which was not originally planned. However, it combined all research into one practical example of how to create such a cloud shader.
Compared to state-of-the-art cloud shaders, the one made during this project can keep up in terms of appearance and customization options, but lacks good performance and the ability to render different cloud types.
In section 4, sphere tracing is extensively described and explained in detail, but was not used in the final prototype, as there is no defined SDF for a cloud volume. The same applies for all shadow related algorithms. The shader does not use shadow rays nor include ambient occlusion.

8 Project Management

8.1 Schedule Comparison

The following chart shows the original schedule (in grey) with a side-by-side comparison with the actual time spent for each task (in blue). It indicates that the schedule was mostly met throughout the project.



As explained in subsection 8.2, "prototype 3" was removed from the schedule, which is why the blue milestone is missing. Since there was now more time available for the other prototype, the milestone for "prototype 2" was moved to the end of the segment.

8.2 Goal Discrepancies

Originally, the following three prototypes were planned.

- volumetric rendering
- procedural noise generation
- ray marching

During research and prototyping, it came clear that "ray marching" is in fact a substantial part of volumetric rendering instead of a completely different topic. Hence, only two of the three listed prototypes were implemented. This change led to a significant boost in available development time for the other two prototypes, for which the results could now be finalized to a greater extent.

8.3 Future Work

The final prototype is a solid basis for future projects, such as rendering of more cloud genera like the infamous, high-towering cumulonimbus clouds, performance optimizations and much more.

8.3.1 Complete Weather System

Another wonderful idea is to expand the shader into a fully-fledged weather system. Instead of having all the those technical parameters, it would instead be dependent on temperature, humidity, altitude, highs and lows, weather fronts and many other real meteorological variables. It would automatically start to rain when the the conditions are met and the cloud shader movement would adjust itself by the wind strength and direction parameters.

8.3.2 Extensive Lighting Features

Despite the considerable effort already put into lighting and illumination methods, there are still some features missing. One of those are *god rays*, the volumetric light shafts that shine through gaps in clouds, giving the scene even more depth. Other absent features are the sun's and moon's halo: A bright circle around the celestial body.

8.3.3 Measure Realism

As described in subsection 6.6, there are several possible approaches to measure the realism of the clouds. This opens up potential for a future project.

8.4 Project Conclusion

It is noteworthy that two more weeks were put into research. This is mainly because new methods and algorithms have been continuously researched during prototyping, which resulted in constant documentation of those findings. However, this did not conflict with the rest of the schedule.
Still, the total amount of time spent was about ten percent more than the originally estimated time budget of 128 hours. This is probably due to the fact that there was quite some effort put into the final prototype.
The project occurred during the same time as the global coronavirus lockdown phase, but was unhindered by that.

In summary, all project requirements were met, the milestones were completed in time and the final prototype turned out great, making this a successful and very informative project.

Glossary

Ambient occlusion Also known as contact shadows, this method darkens points in the scene that are not or only slightly exposed to the light and its environment. 17, 44

Axis-aligned bounding box A non-rotated bounding box enclosing an object completely. 29

Billboard A 2D image always facing towards the main camera. 3

Compute shader A shader which runs on the GPU but outside of the default render pipeline. 43

Constructive solid geometry Short CSG, stands for combining primitive geometric objects with Boolean operators. 16

Convection Convection describes the transfer of heat from movement of liquid or gas. 2

Convolutional neural network A neural network that is able to classify images. 42

Fractal noise In this matter, the same as fractal Brownian motion. 26, 27

Fractal Brownian motion Different iterations of continuously more detailed noise layered on top of each other. i, 26, 47

Generative adverserial network A set of two neural networks, where one generates images and the other tries to tell wether those images are real or generated. 42

GPU Graphics Processing Unit. 43

Gradient The *gradient* denotes the direction of the greatest change of a scalar function. 11

Histogram A graphical representation of data like brightness or color distribution of a given photograph. 42

Lightmarching The same concept as ray marching, but instead of being cast into the volume, it is cast towards the primary light source with a constant step. 34, 35, 36, 40, 43, 52

Low poly A 3D polymesh with a relatively low count of polygons. 4

Noise A randomly generated pattern, referring to procedural pattern generation. i, 20, 45

Noise generation Noise generation is used to generate textures of one or more dimension with seemingly random smooth transitions from black to white (zero to one). 18

Parameters Shader variables exposed to the Unity Editor. 28, 31, 33, 38, 41, 46

Penumbra The partially shaded outer region of diffuse shadows. Also described as soft edges. 14

Polymesh A polymesh is a 3D model composed of polygons or triangles. 4, 11, 51

Procedural Created solely with algorithms and independant of any prerequisites. i, 19, 20, 45, 47

Ray marching Ray marching is a type of method to approximate the surface distance of a volumetric object, where a ray is cast into the volume and stepped forward until the surface is reached. i, 6, 7, 8, 13, 28, 34, 35, 43, 45, 47

Scalar field A scalar field describes a typically three-dimensional grid of elements called *voxels*, each containing a scalar value. 6

Shape blending In SDFs, shapes can be seemingly blended together by returning a interpolated value of those distances. 15

Signed distance function A signed distance function, short SDF, returns a positive distance if the origin is outside the volume and a negative distance if it is inside the volume. 9, 11, 15, 16, 29

Sphere tracing Sphere tracing describes an optimized algorithm of ray marching by using signed distance functions to approximate the surface distance of the volume. 9, 44

Subsurface scattering SSS is a mechanism of light transport in which light enters a translucent object, is scattered around and exits the material at a different point, resulting in illuminated areas where the material is thin. 32

Sunlight forward scattering The process of sunlight shining through and illuminating the clouds which cover the sun. 32, 40, 48

Sunlight transmittance In this matter, the same as sunlight forward scattering. 32, 41

Surface normal A *surface normal* or *normal* is a vector which is perpendicular to a given geometry, like a triangle or polygon. i, 11, 12, 34

Translucent An object or substance that is translucent allows light to be passed through it, meaning it is rendered transparently to some degree. 29

Vector field It is the same as a scalar field, except the voxels are vector values. 6

Volumetric rendering This describes a technique which takes a 3D volume of data and projects it to 2D. It is mostly used for transparent effects stored as a 3D image. i, 6, 11, 28, 29, 45

Voxel Short for *volume element*, a voxel is a value (either a number or a vector) on a scalar or vector field . 6

World space Coordinates defined with respect to a global Cartesian coordinate system. 3

References

[1] Andrew Schneider, *Real-time volumetric cloudscapes*, https://books.google.ch/books?hl=en&lr=&id=rA7YCwAAQBAJ&oi=fnd&pg=PA97&dq=real+time+rendering+of+volumetric+clouds&ots=tu16W0NkOZ, [Online; accessed April 2, 2020], 2016.

[2] Jean-Philippe Grenier, *Volumetric clouds*, https://area.autodesk.com/blogs/game-dev-blog/volumetric-clouds/, [Online; accessed April 2, 2020], 2016.

[3] Jamie Wong, *Ray marching and signed distance functions*, http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/, [Online; accessed April 3, 2020], 2016.

[4] Alan Zucconi, *Volumetric rendering*, https://www.alanzucconi.com/2016/07/01/volumetric-rendering/, [Online; accessed April 3, 2020], 2016.

[5] Patricio Gonzalez Vivo, Jen Lowe, *The book of shaders*, https://thebookofshaders.com/, [Online; accessed April 14, 2020], 2015.

[6] Sebastian Lague, *Lague's youtube channel about shader coding*. [Online]. Available: https://www.youtube.com/user/Cercopithecan.

[7] *Photographic reference of stratus clouds*. [Online]. Available: https://en.wikipedia.org/wiki/Stratus_cloud.

[8] *Photographic reference of cirrus clouds*. [Online]. Available: https://en.wikipedia.org/wiki/Cirrus_cloud.

[9] *Photographic reference of an altocumulus cloud formation*. [Online]. Available: https://en.wikipedia.org/wiki/Altocumulus_cloud.

[10] *Photographic reference of stratocumulus cloudscape*. [Online]. Available: http://www.randotunisie.tn/wp-content/uploads/2018/06/.

[11] *A polymesh of a cloud*. [Online]. Available: https://www.utilitydesign.co.uk/magis-metal-mesh-clouds.

[12] *Several volumetric cloudscapes from the game horizon: Zero dawn, drawn in real time*. [Online]. Available: https://tech4gamers.com/horizon-zero-dawn-gets-new-screenshots/.

[13] Joey de Vries, *Phong illumination model*, https://learnopengl.com/Lighting/Basic-Lighting, [Online; accessed April 8, 2020], 2014.

[14] Ken Perlin, *Improved noise reference implementation*, https://mrl.nyu.edu/~perlin/noise/, [Online; accessed April 27, 2020], 2002.

[15] ——, *Improving noise*, https://mrl.nyu.edu/~perlin/paper445.pdf, [Online; accessed May 04, 2020], 2002.

[16] Patricio Gonzalez Vivo, Jen Lowe, *Fractal brownian motion*, https://thebookofshaders.com/13/, [Online; accessed May 05, 2020], 2015.

[17] A. Majercik, C. Crassin, P. Shirley, and M. McGuire, *A ray-box intersection algorithm and efficient dynamic voxel rendering*, http://jcgt.org/published/0007/03/04/, 2018.

[18] *Different cloud types*. [Online]. Available: https://www.gotoknow.org/posts/624816.

[19] *Cloudy night sky video*. [Online]. Available: https://videohive.net/item/clouds-night-sky/15135126.

[20] *Altostratus clouds at sunset.* [Online]. Available: https://www.pinterest.ch/pin/487725834617556668/.

[21] *Cloudy and rainy weather report images.* [Online]. Available: https://www.ekathimerini.com/238896/article/ekathimerini/news/weather-to-turn-cloudy-and-rainy-on-tuesday.

List of Figures

Listings

Glossary

Altitude A vertical distance measurement, in this context specifically the distance from sea level to the given object. 2, 5, 6, 11

Cloudlet Small, white, puffy clouds that come in large quantities, together forming a cloud of the cumulus family. 6, 7, 37

CNN A neural network that is able to classify images. 41

Cold front A cold weather front, the boundary of a mass of air that carries cold or cool air. When colliding with a warm front, precipitation is often followed. 3, 4, 9, 130, 133

Compute shader A shader which runs on the GPU but outside of the default render pipeline. 26, 27, 29, 36, 43, 47, 49, 129, 135

Convection The process of warm air rising from the surface and cooling at higher altitude, of which the moisture is then condensed into clouds. 2, 7, 8, 9

Desublimation The process of gas transitioning to liquid without passing through the liquid phase. 6

FPS Frames per second, a measurement of how fast the application is performing (60 is good). 43, 45

Fractal Brownian motion Different iterations of continuously more detailed noise layered on top of each other. 19, 29

Fragment In computer graphics, a fragment is a single pixel on the screen that is processed by a fragment shader and given a color in the process, effectively rendering it. 128

Fragment shader A shader that processes single pixels, called fragments, calculates its color and outputs that to the frame buffer. 29, 128

Frame rate The rate at which a new image (called frame) appears on the display. 49

Frame buffer The buffer that stores pixels for each frame, from which the monitor constantly reads. The monitor then displays those pixels on the screen. 26, 128

GAN A set of two neural networks, where one generates images and the other tries to tell wether those images are real or generated. 41

GPU Graphics processing unit. A piece of hardware designed to rapidly manipulate and alter memory, often intented for output to a display device. 27, 130

Halo phenomenon White or colored rings or arcs of light around the sun or the moon, produced by cirrostratus clouds. 6

HDRP Unityś render workflow for high fidelity, high quality projects. 34

Histogram A graphical representation of data like brightness or color distribution of a given photograph. 41

HLSL High-level shading language. Developed by microsoft, this is a standard shader language for DirectX used in graphics programming. 21, 24, 26

In-engine In computer graphics, this refers to being inside the game engine; being measured or rendered by the game engine. 36, 41

Kernel In compute shaders, the kernel represents an entry point and defines the method that is executed for each thread group when running the compute shader. 26, 27, 28, 29

Light marching The same concept as ray marching, but instead of being cast into the volume, it is cast towards the primary light source with a constant step. 36, 43, 49

Linear interpolation Simply put, linear interpolation describes a method of finding values inbetween two points on the same line. 17, 18

Neural network A series of algorithms that can recognize and categorize certain patterns in a given set of data. 40, 128

Noise A randomly generated pattern, referring to procedural pattern generation. 19, 20, 21, 22, 23, 24, 25, 29, 35, 36, 38, 40, 43, 133, 134, 135

Noise generation Noise generation is used to generate textures of one or more dimension with seemingly random smooth transitions from black to white (zero to one). i, 19, 22, 36, 49

Occluded front When a cold front overtakes a warm front, it pushes the warm air upwards (thermals). The moisture of the warm air condenses as it rises, creating water vapor. This often results in clouds with precipitation. 4, 7, 8, 129, 133

Occlusion In meteorology, the clash of a warm front and a cold front. See occluded front. 4, 133

Particle system In computer graphics, a particle system is a technique that continuously spawns and recycles objects. They are often used to reproduce fire or smoke effects, with small flame or dust textures as particles. 18, 133

Post-processing The act of applying additional effects to a rendered image before displaying it on the monitor. 34, 39

Precipitation Rainfall. The result of atmospheric water vapor that has been condensed and now falls from clouds. 3, 4, 6, 7, 8, 10, 14, 38, 128, 129, 130, 133

Procedural Created solely with algorithms and independant of any prerequisites. i, 129

Proxy plane A proxy object that is a plane. 34, 39, 134

Proxy object Regarding this project, proxy objects are game objects that substitute a certain visual effect like the halo of the sun or the darkening of the sky. 33, 34, 39, 129

Pseudo-random A random number generated with a deterministic algorithm, meaning that the same input will always give the same output. 19, 20, 133

Rasterization Rasterization describes the final step in rendering. It is the task of taking an image described in vector geometry and converting it into a raster image (a series of pixels). 26

Ray marching Ray marching is a type of method to approximate the surface distance of a volumetric object, where a ray is cast into the volume and stepped forward until the surface is reached. i, 36, 43, 49, 129

ROP The render output pipeline, a component responsible for calculating the final pixel colors or depth values via specific matrix and vector operations. 26

Shader A piece of software which runs on the GPU, rendering geometrically defined objects to the screen. 12, 13, 15, 17, 18, 26, 27, 29, 32, 33, 36, 40, 128, 130

Shadow pass A second shader pass that only calculates the shadow of its object. 36, 134

Sunlight forward scattering The process of sunlight shining through and illuminating the clouds which cover the sun. 36

Texel Short for texture element, a single pixel of a 2D texture. 26, 36

Texture slice A 2D texture extracted from a 3D texture for a given depth. 22, 24, 25, 133, 134

Thermal In relation with meteorology, the hot, rising air from convection is called "thermal". 2, 3, 4, 129, 133

UI User interface. The interface that allows the user to interact with the software. 32, 42, 43, 44, 45, 50

Update loop The process that updates all components every frame, like updating the scene view and game objects. 35

Volumetric This describes a technique which takes a 3D volume of data and projects it to 2D. It is mostly used for transparent effects stored as a 3D image. 32, 33, 36, 43

Warm front A warm weather front, the boundary of a mass of air that carries mild or warm air. When colliding with a cold front, precipitation is often followed. 3, 4, 6, 128, 133

Water vapor Evaporated water in a gaseous form. 6, 129

Weather rendering system The Unity application that is implemented during this project. It takes in live data from a weather service and uses topological elevation models to create a weather simulation, which is then rendered and up for comparison with live photographs. 5

Weather front A boundary between to air masses, which differ in temperature, wind direction and humidity. 2, 128, 130

WMO A specialized agency conducting atmospheric science, climatology, hydrology and geophysics. 5

References

[1] *Lifting by convection*, [Online; accessed April 08, 2021], 2010. [Online]. Available: http://ww2010.atmos.uiuc.edu/(Gh)/wwhlpr/convection.rxml?hret=/guides/mtr/cld/cldtyp/mdl/altcu.rxml.

[2] *Metoffice: Weather fronts*, [Online; accessed April 08, 2021]. [Online]. Available: https://www.metoffice.gov.uk/weather/learn-about/weather/atmosphere/weather-fronts.

[3] *Ww2010: Precipitation along a warm front*, [Online; accessed April 09, 2021]. [Online]. Available: http://ww2010.atmos.uiuc.edu/(Gh)/wwhlpr/warm_front_precip.rxml?hret=/guides/mtr/cld/cldtyp/mdl/altcu.rxml&prv=1.

[4] *Ww2010: Precipitation along a cold front*, [Online; accessed April 09, 2021]. [Online]. Available: http://ww2010.atmos.uiuc.edu/(Gh)/wwhlpr/cold_front_precip.rxml?hret=/guides/mtr/cld/cldtyp/mdl/altcu.rxml.

[5] *Skybrary: Occluded fronts*, [Online; accessed April 11, 2021]. [Online]. Available: https://www.skybrary.aero/index.php/Occluded_Front.

[6] *List of cloud types*, [Modified; Online; accessed April 08, 2021], 2006. [Online]. Available: https://en.wikipedia.org/wiki/List_of_cloud_types.

[7] *Wikipedia: Cirrus clouds*, [Online; accessed April 08, 2021], 2006. [Online]. Available: https://en.wikipedia.org/wiki/Cirrus_cloud.

[8] *How to predict the weather using clouds*, [Online; accessed April 08, 2021], 2020. [Online]. Available: https://www.countryfile.com/how-to/outdoor-skills/how-to-predict-the-weather-forecast-using-clouds/.

[9] *Wikipedia: Cirrostratus clouds*, [Online; accessed April 08, 2021]. [Online]. Available: https://www.metoffice.gov.uk/weather/learn-about/weather/types-of-weather/clouds/high-clouds/cirrostratus.

[10] *Meteoblue: Cloud types*, [Photos by Julian Gutbrod; Online; accessed April 08, 2021]. [Online]. Available: https://content.meteoblue.com/en/meteoscool/weather/clouds/cloud-types.

[11] *Wikipedia: Cirrocumulus clouds*, [Online; accessed April 08, 2021]. [Online]. Available: https://en.wikipedia.org/wiki/Cirrocumulus_cloud.

[12] *Wikipedia: Altocumulus clouds*, [Online; accessed April 08, 2021]. [Online]. Available: https://www.metoffice.gov.uk/weather/learn-about/weather/types-of-weather/clouds/mid-level-clouds/altocumulus.

[13] *Wikipedia: Nimbostratus clouds*, [Online; accessed April 08, 2021]. [Online]. Available: https://en.wikipedia.org/wiki/Nimbostratus_cloud.

[14] *Wikipedia: Cumulonimbus clouds*, [Photos by Julian Gutbrod; Online; accessed April 08, 2021]. [Online]. Available: https://en.wikipedia.org/wiki/Cumulonimbus_cloud.

[15] *Metoffice: Cumulonimbus clouds*, [Online; accessed April 11, 2021]. [Online]. Available: https://www.metoffice.gov.uk/weather/learn-about/weather/types-of-weather/clouds/low-level-clouds/cumulonimbus.

[16] P. G. Vivo and J. Lowe, *Book of shaders: Noise*, [Online; accessed April 30, 2021], 2015. [Online]. Available: https://thebookofshaders.com/11/.

[17] M. Thomann, *Project 2 - procedural cloud shader*, [Section 5: Noise Generation], 2020.

[18] ——, *Project 2 - procedural cloud shader*, [Subsection 5.1: Random Numbers], 2020.

[19] ——, *Project 2 - procedural cloud shader*, [Subsection 5.3.1: Perlin Noise], 2020.

[20] ——, *Project 2 - procedural cloud shader*, [Subsection 5.3.3: Fractal Brownian Motion], 2020.

[21] S. Worley, *A cellular texture basis function*, [Online; accessed April 30, 2021], 1996. [Online]. Available: `http://www.rhythmiccanvas.com/research/papers/worley.pdf`.

[22] *Ronja tutorials: Tiling noise*, [Online; accessed April 30, 2021], 2018. [Online]. Available: `https://www.ronja-tutorials.com/post/029-tiling-noise/`.

[23] *Opengl: Mod declaration*, [Online; accessed April 30, 2021], 2014. [Online]. Available: `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/mod.xhtml`.

[24] *Arcgis: Maps sdk for unity*, 2021. [Online]. Available: `https://developers.arcgis.com/unity-sdk/`.

[25] P. Rademacher, J. Lengyel, E. Cutrell, and T. Whitted, *Measuring the perception of visual realism in images*, [Online; accessed June 06, 2021], 2001. [Online]. Available: `https://diglib.eg.org/xmlui/bitstream/handle/10.2312/EGWR.EGWR01.235-248/235-248.pdf`.

[26] ——, *Measuring the perception of visual realism in images*, [Section 6.3: Results], 2001. [Online]. Available: `https://diglib.eg.org/xmlui/bitstream/handle/10.2312/EGWR.EGWR01.235-248/235-248.pdf`.

[27] *Meteoblue weather in new microsoft flight simulator 2020*, [Online; accessed June 07, 2021], 2020. [Online]. Available: `https://www.meteoblue.com/en/blog/article/show/39819_meteoblue+weather+in+new+Microsoft+Flight+Simulator2020`.

List of Figures

Listings