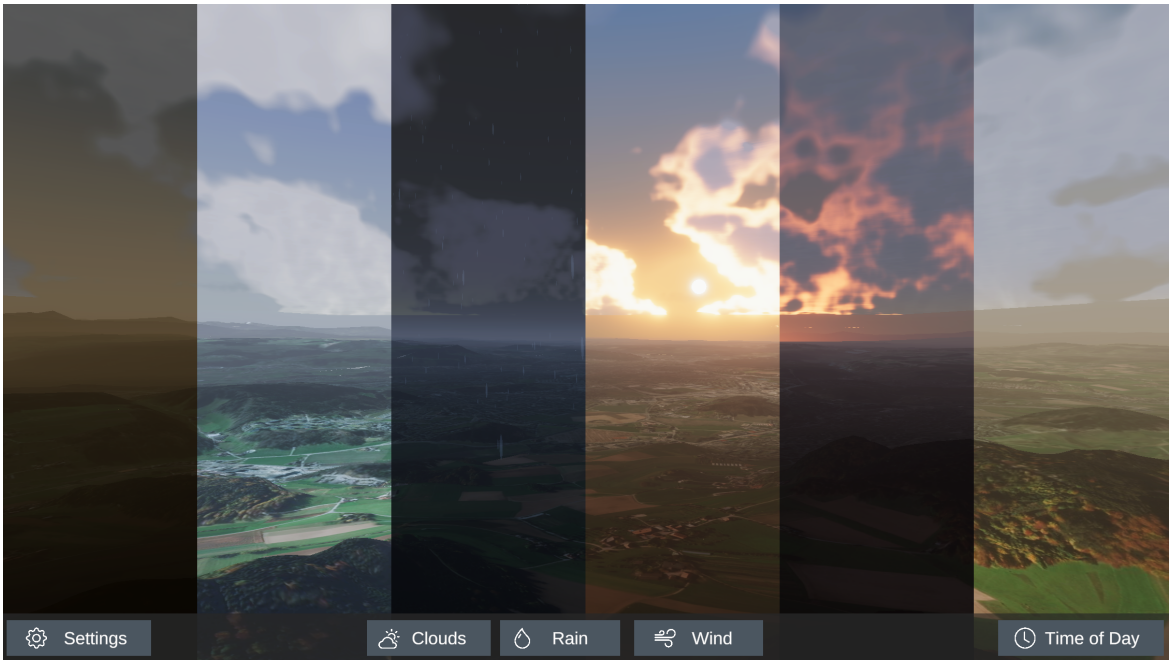
Near Real-time Weather Rendering System

Project documentation



Field of Studies:	BSc in Computer Science
Specialization:	Computer perception and virtual reality
Author:	Matthias Thomann
Supervisor:	Prof. Urs Künzler
Date:	June 7, 2021
Version:	1.0

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 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

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].

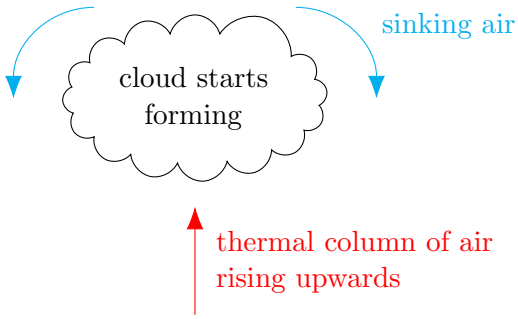


Figure 1: Lifting by convection.

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 temparature, 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].

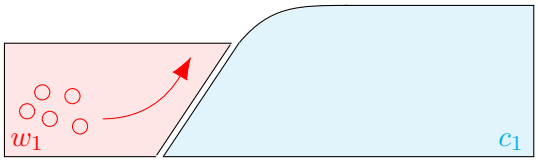


Figure 2: Warm front: warmer air advances, rising over the colder air, cooling down in the process.

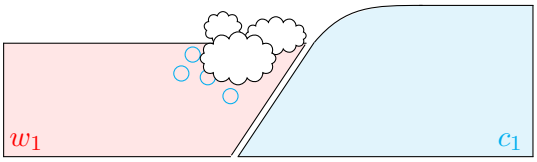


Figure 3: Warm front: as the air cools down, the moisture condenses. Clouds start to form.

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].

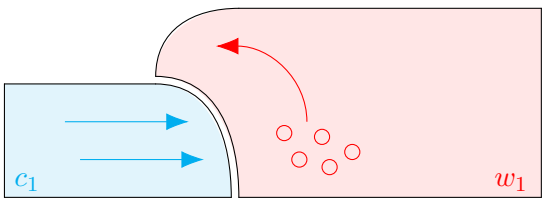


Figure 4: Cold front: colder air advances, pushing the warmer air upwards, cooling it down.

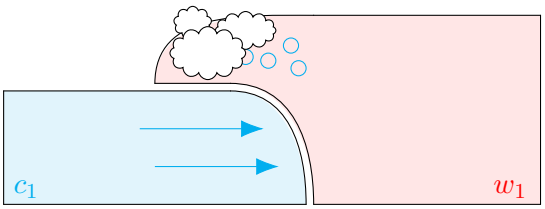


Figure 5: Cold front: as the air cools down, the moisture condenses. Clouds start to form.

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].

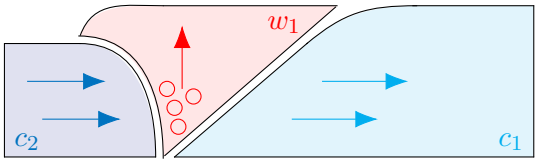


Figure 6: Cold occlusion: cool air catches up with a preceding cold front, forcing the warmer air inbetween to go up, creating a thermal.

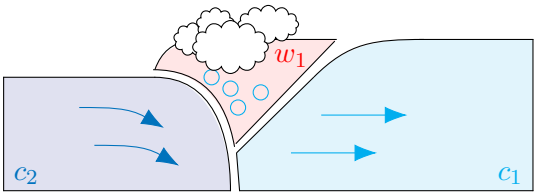


Figure 7: Cold occlusion: the cool air pushes underneath both other fronts. An occluded front is created, bringing heavy precipitation.

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

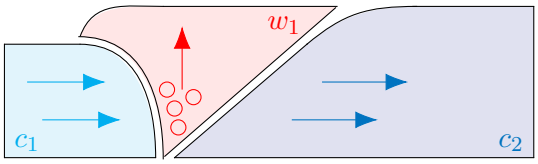


Figure 8: Warm occlusion: a cold front catches up with a warm front preceded by cool air, forcing the warmer air inbetween to go up, creating a thermal.

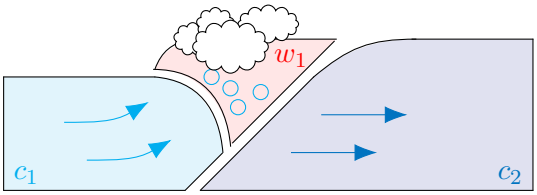


Figure 9: Warm occlusion the cold front is forced to climb over the cool air, pushing the warm front up. An occluded front is created, bringing heavy precipitation.

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.

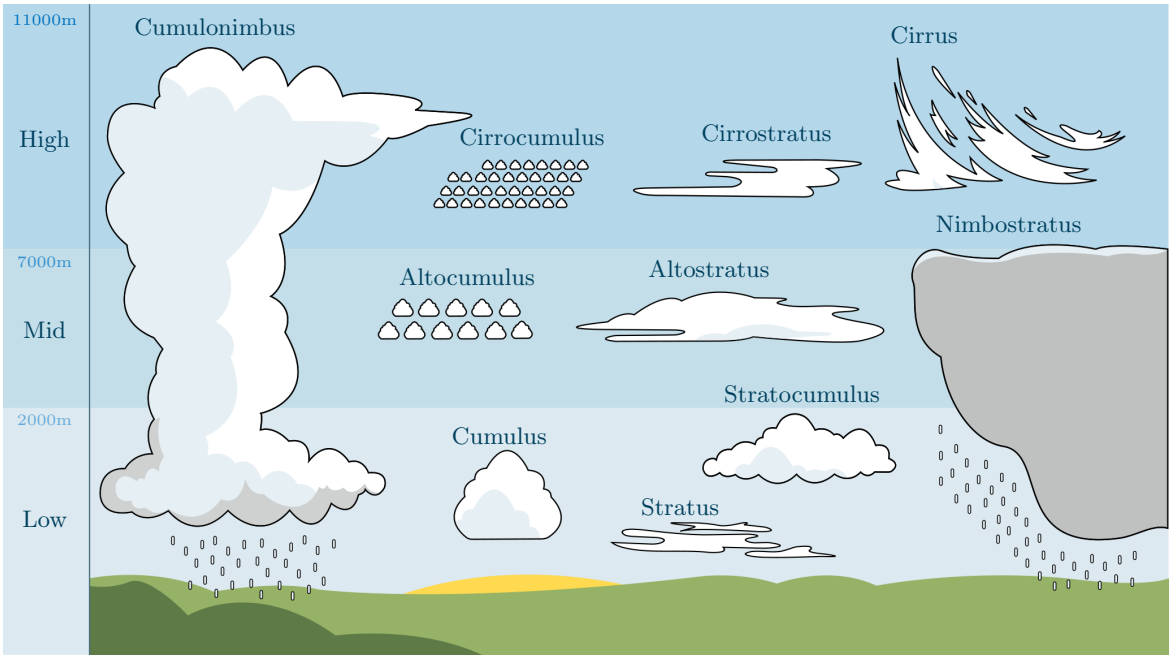


Figure 10: Distinct classifications of cloud shapes in the troposphere [6].

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 wooly look give the cloud the name suffix *cumulus*.
Cirrocumulus clouds are realtively 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 similiarities 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.

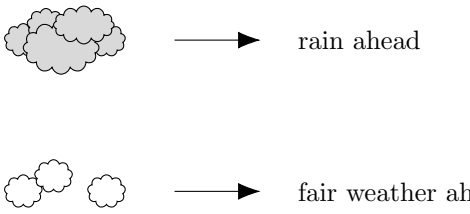


Figure 21: Weather information based on visual data.

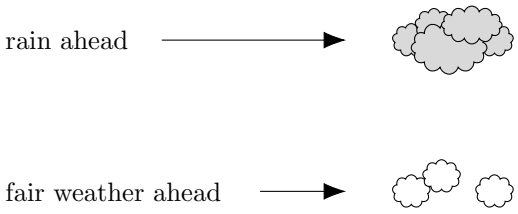


Figure 22: Visual construction based on weather information.

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 troposhpere 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.

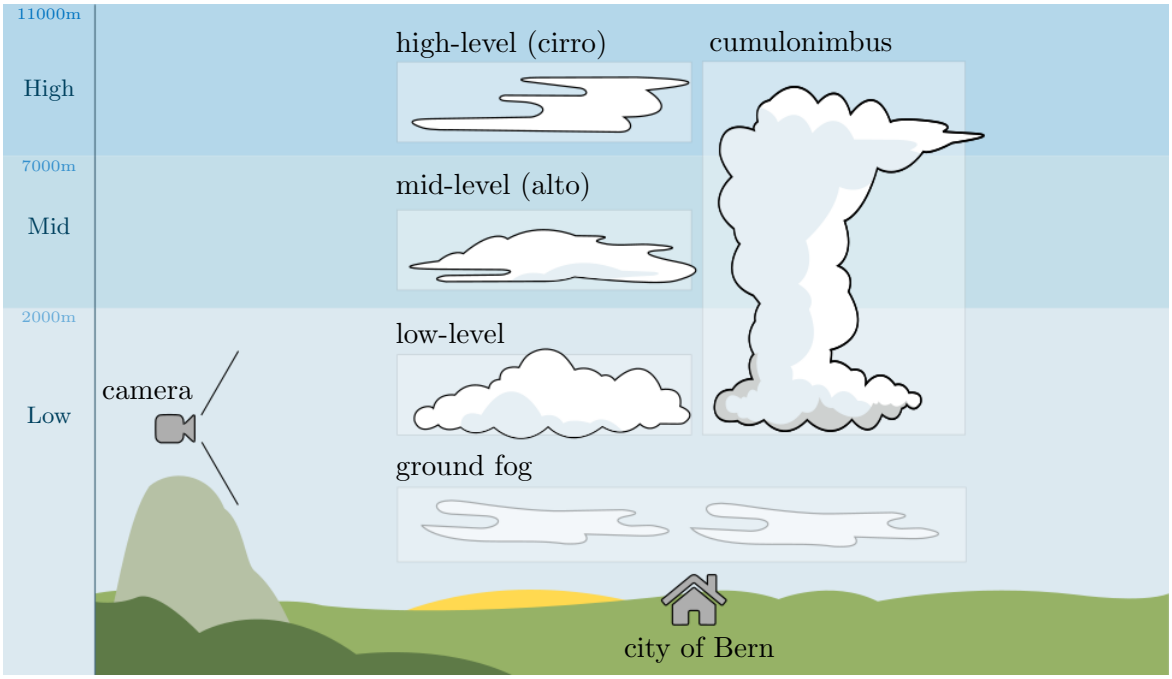


Figure 23: Layers of cloud shaders.

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.

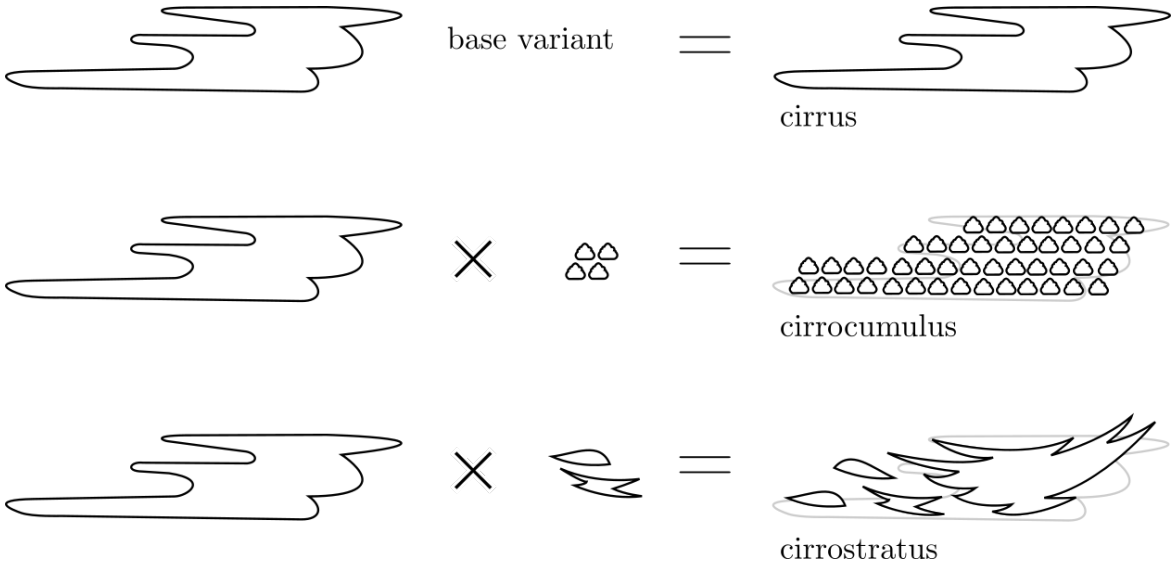


Figure 24: Breakdown of the highest shader layer.

3.2.2 Mid-Level Clouds

The middle layer consists of altostratus and altocumulus clouds, the latter mainly occuring 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 adjustements necessary.

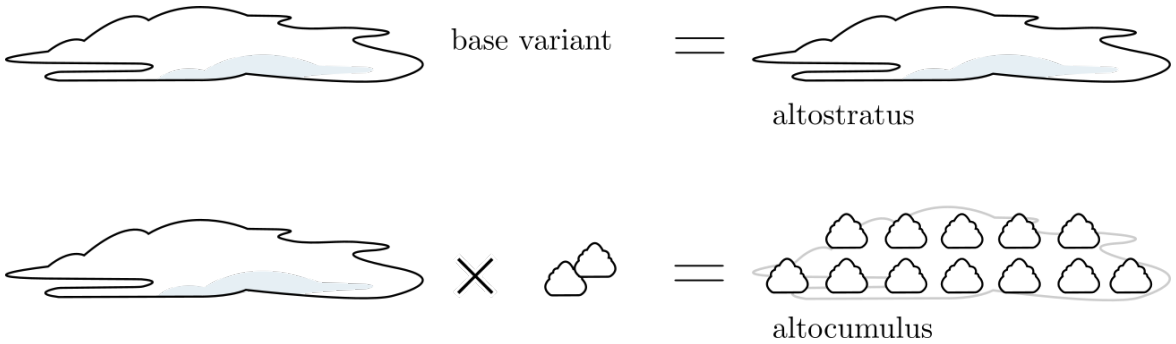


Figure 25: Breakdown of the middle shader layer.

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.

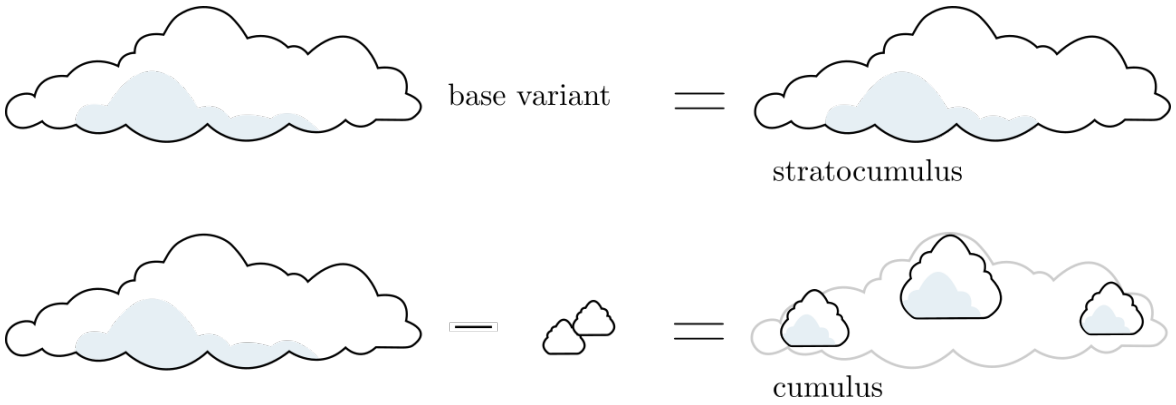


Figure 26: Breakdown of the lowest shader layer.

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.

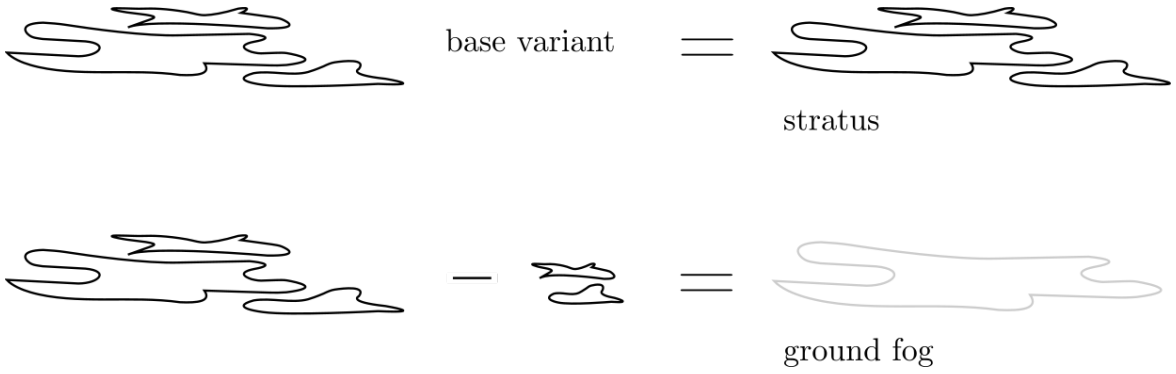


Figure 27: Breakdown of the fog shader layer.

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.

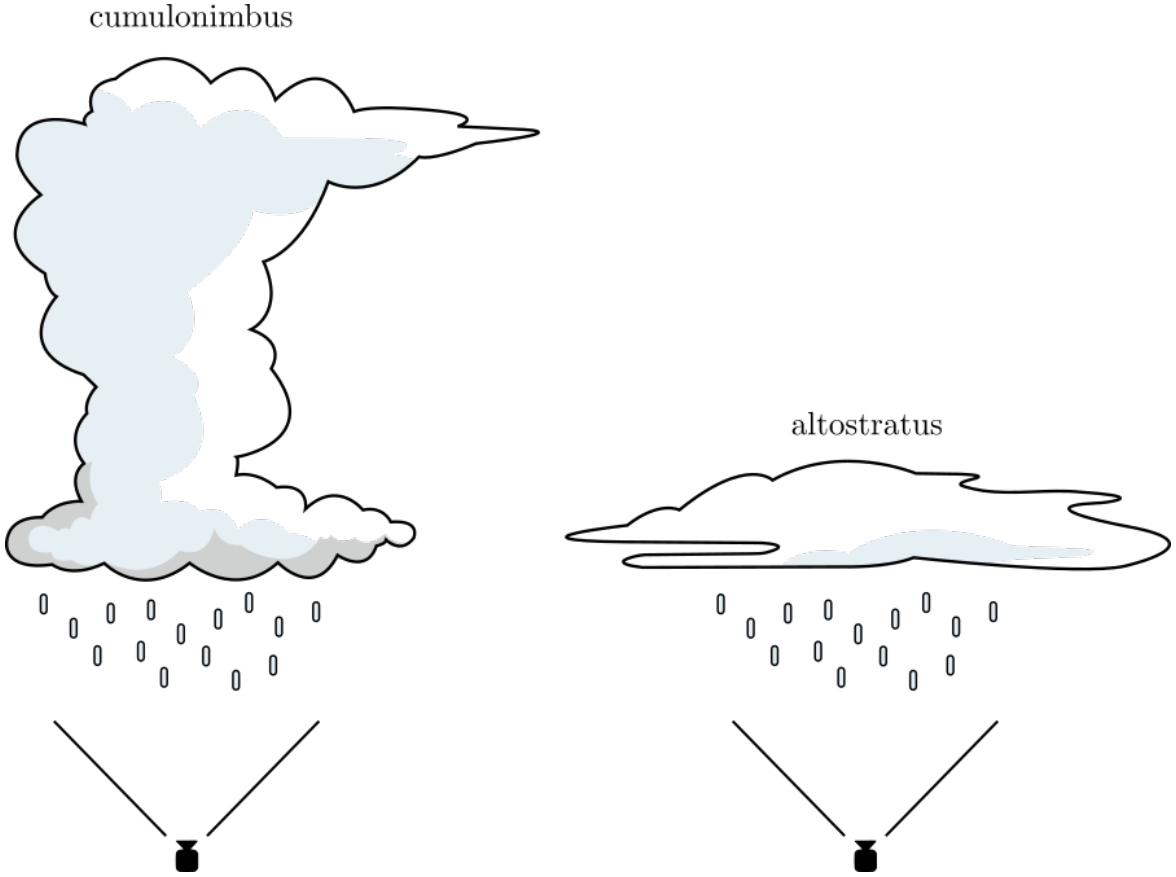


Figure 28: Perspective similarities under clouds with precipitation.

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.

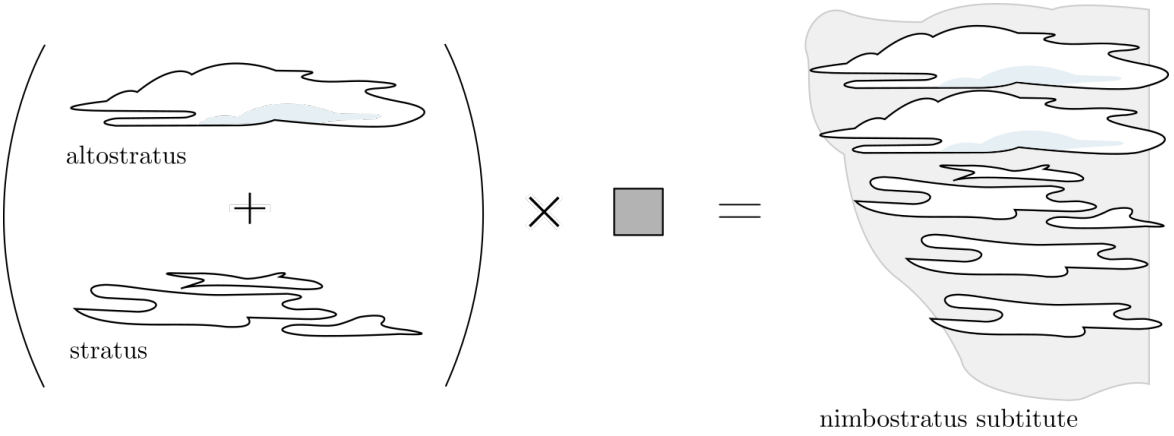


Figure 29: Breakdown of the nimbostratus substitute.

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.

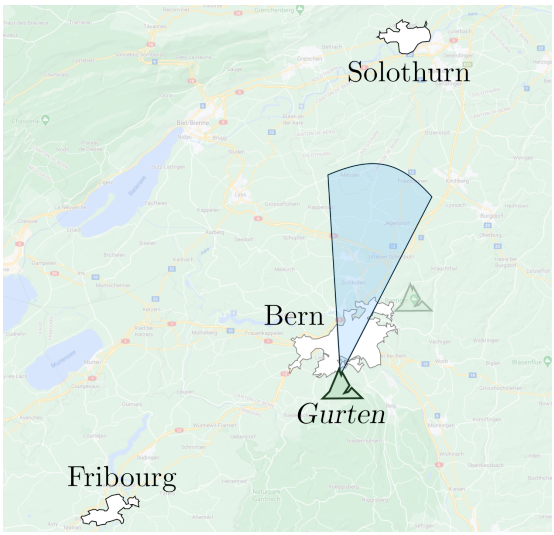


Figure 30: Perspective view from the Gurten mountain: distant weather in Solothurn is considered.

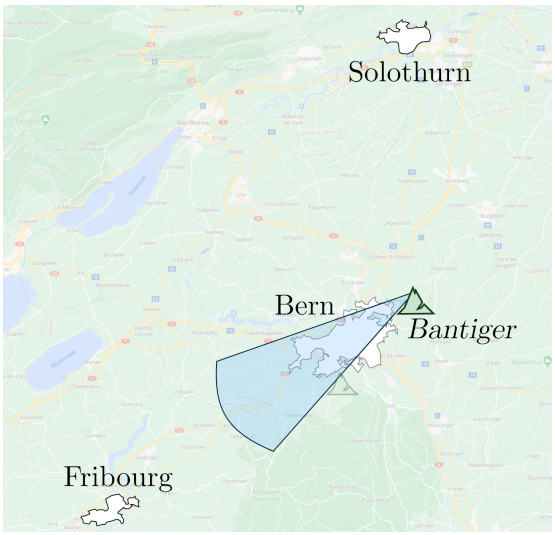


Figure 31: Perspective view from the Bantiger mountain: distant weather in Fribourg is considered.

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 inbetween 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.

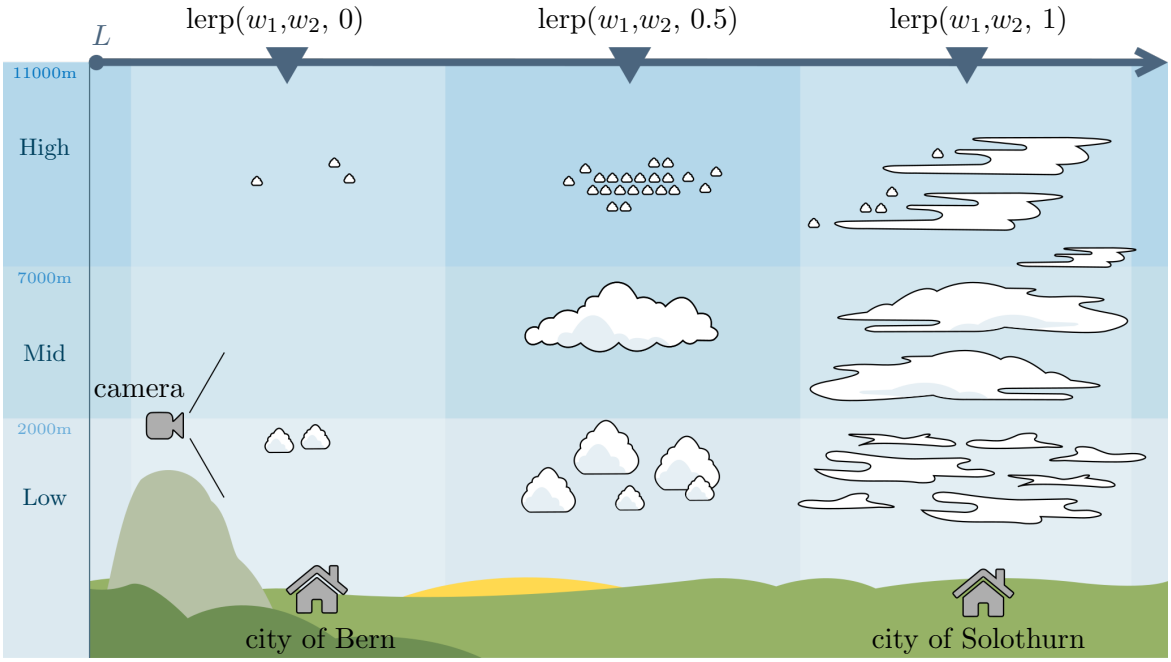


Figure 32: Weather data interpolation for the cloud layers from Bern to Solothurn.

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 inbetween 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.

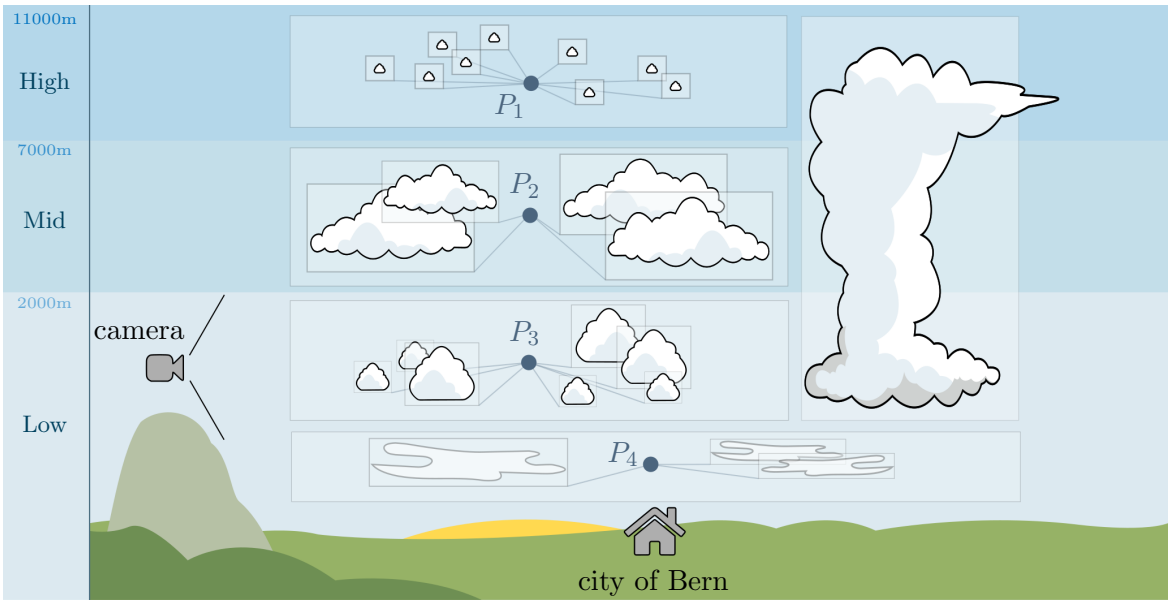


Figure 33: Alternative implementation approach based on particle systems.

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 inbetween 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 phenomenons. For example, the following images show photographs of patterns that seem almost completely random.

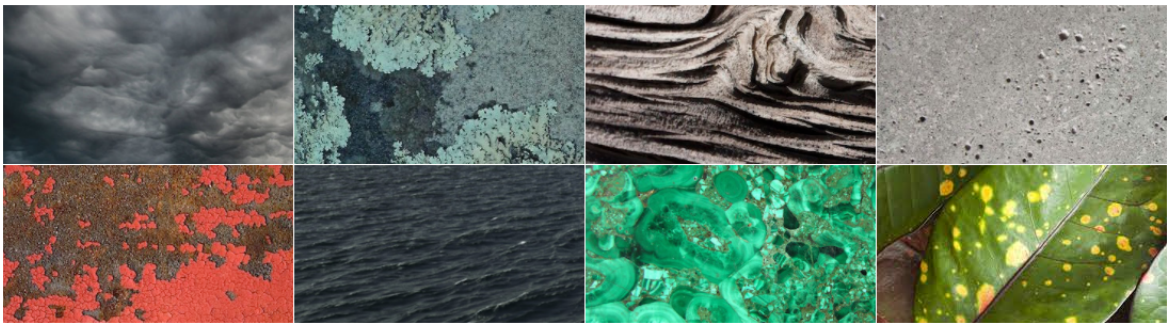


Figure 34: Random patterns observed in Nature [16].

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 thorougly 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.

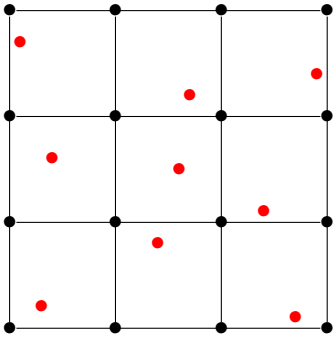


Figure 35: Voronoi grid with pseudo-randomly assigned seed points for each cell.

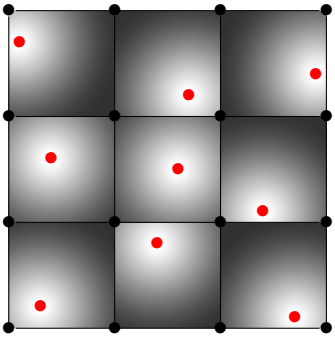


Figure 36: Voronoi grid with seed distances visualized.

As reckognizable 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.

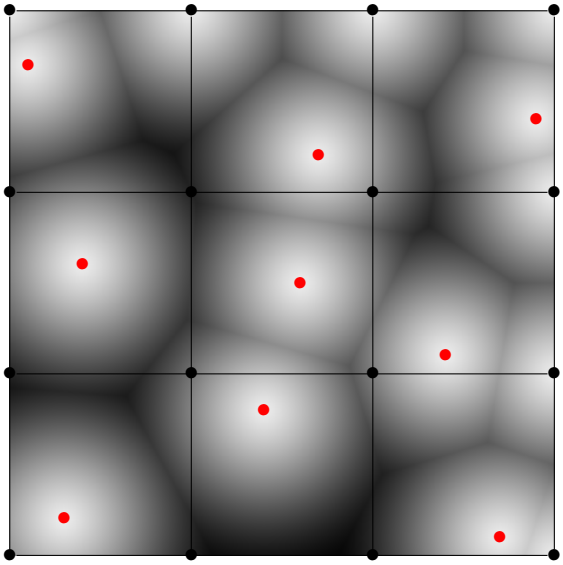


Figure 37: Complete 2D Voronoi noise texture.

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 inbetween the tiles. Tiling the texture saves the calculation of an overly large area of noise and is therefore the desired approach in many cases.

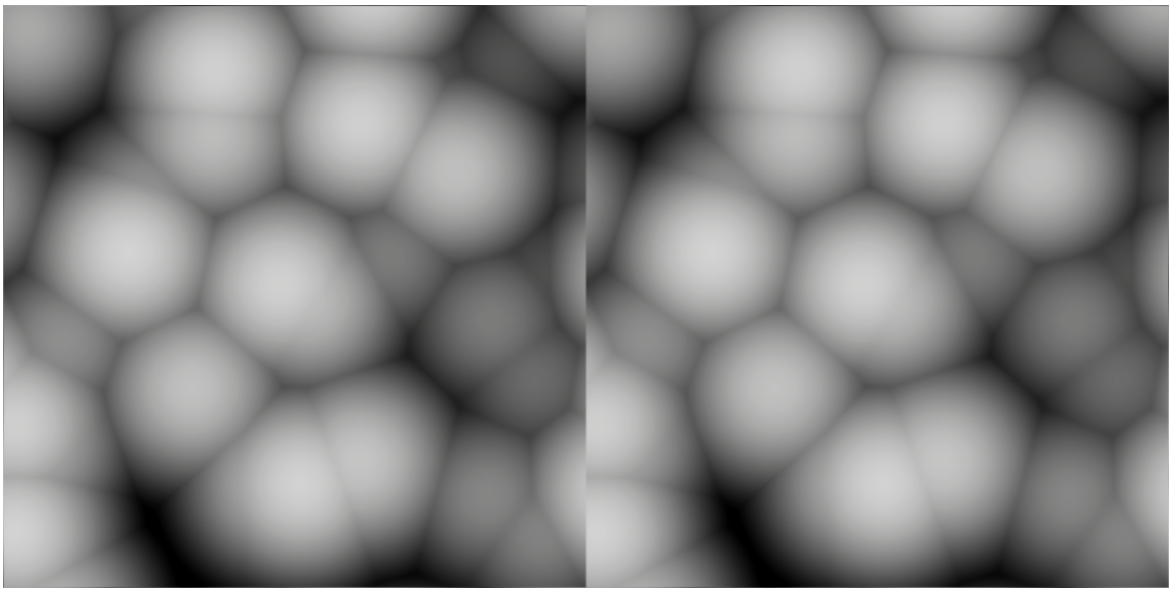


Figure 38: Tiled 3D noise texture slices.

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.

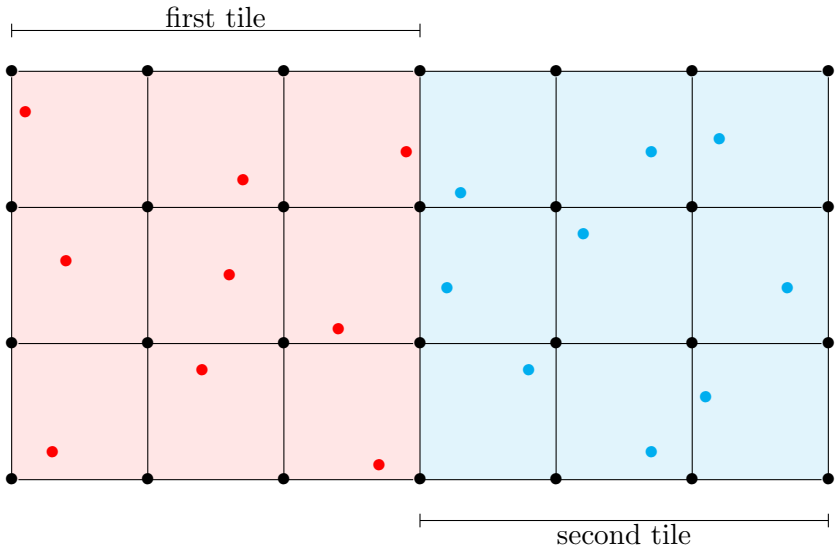


Figure 39: Voronoi noise seeds of the second tile are different than the seeds of the first tile.

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.

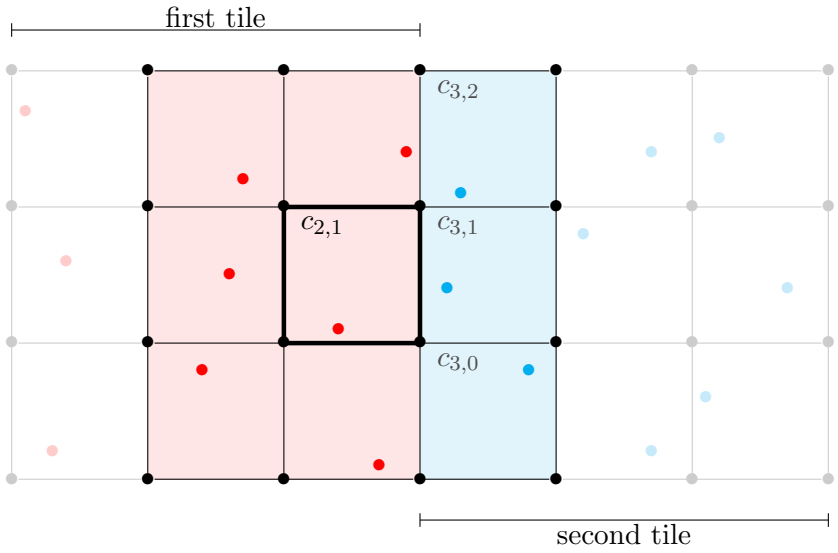


Figure 40: Voronoi noise seeds are sampled from the second tile.

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.

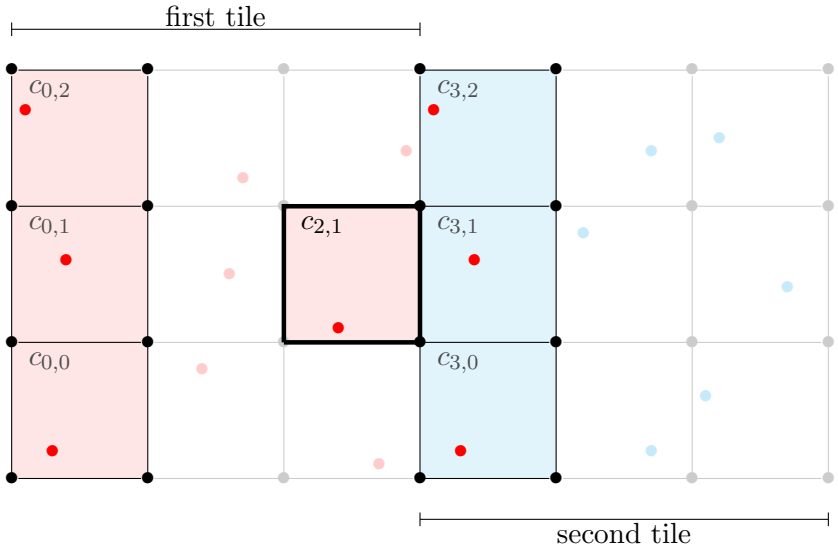


Figure 41: Voronoi noise seeds that exceed the boundary are sampled from the first set again.

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].

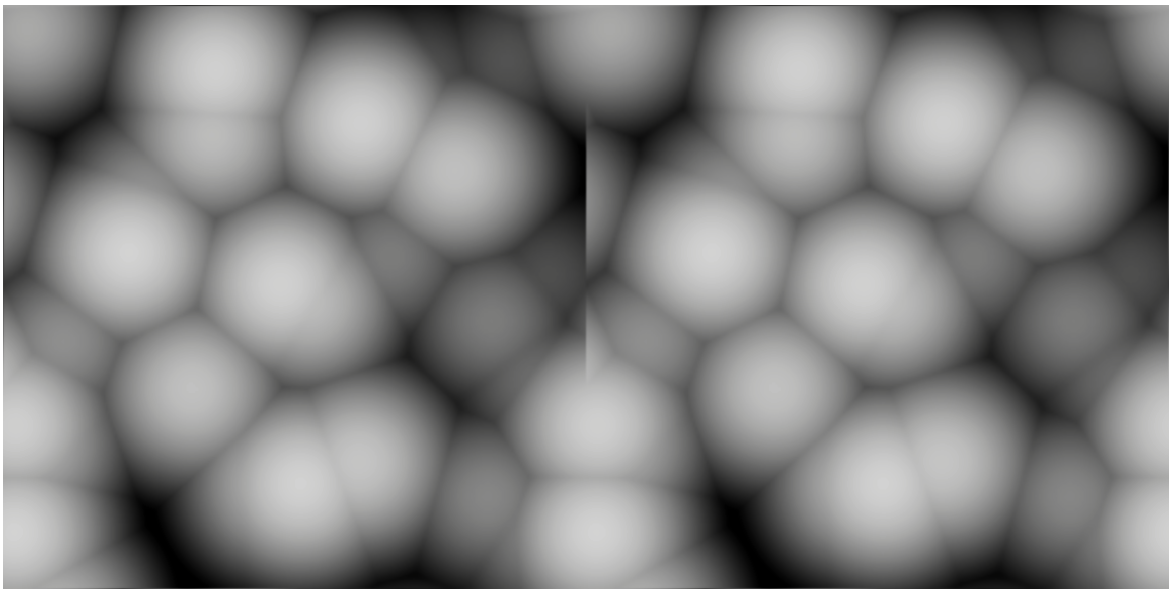


Figure 42: Partially seamless, tiled 3D noise texture slices.

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.

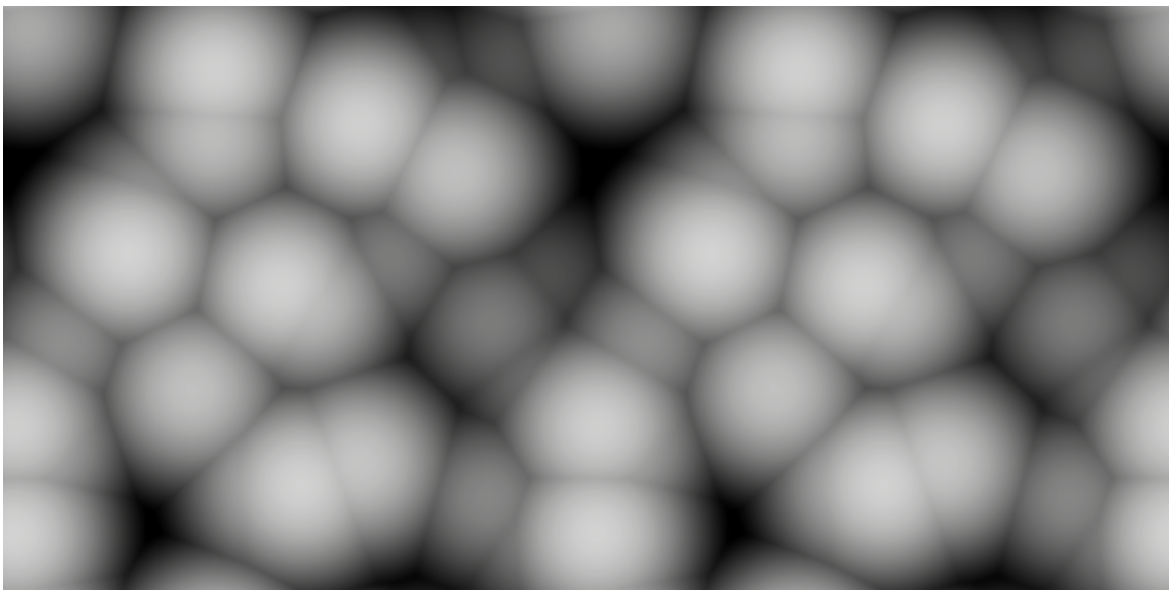


Figure 43: Seamlessly tiled 3D noise texture slices.

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 safed.

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.

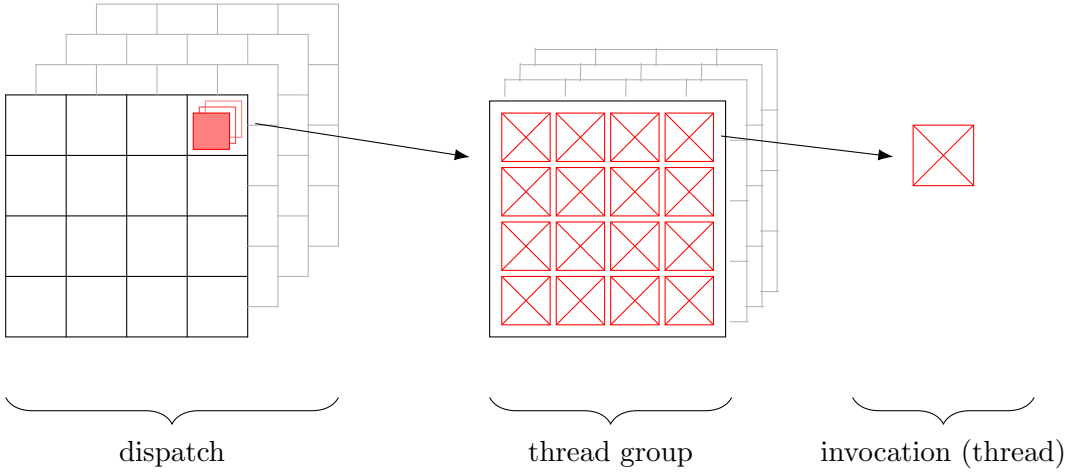


Figure 44: Hierarchical thread structure of compute shaders.

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.

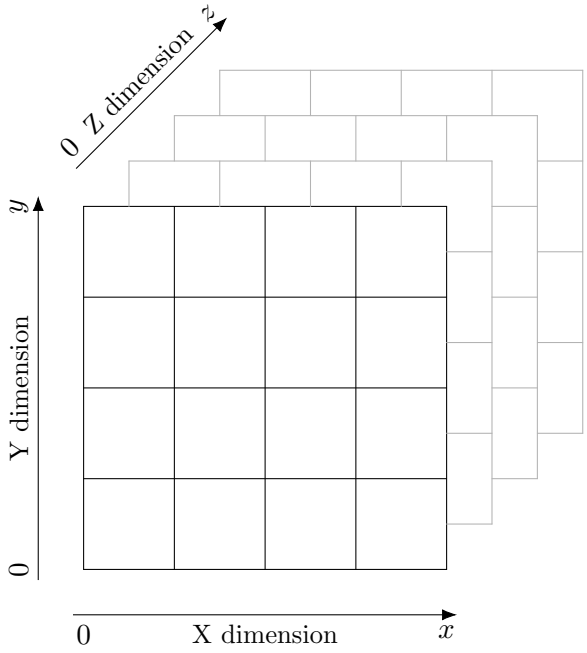


Figure 45: Compute shader thread group with labeled dimensions.

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.

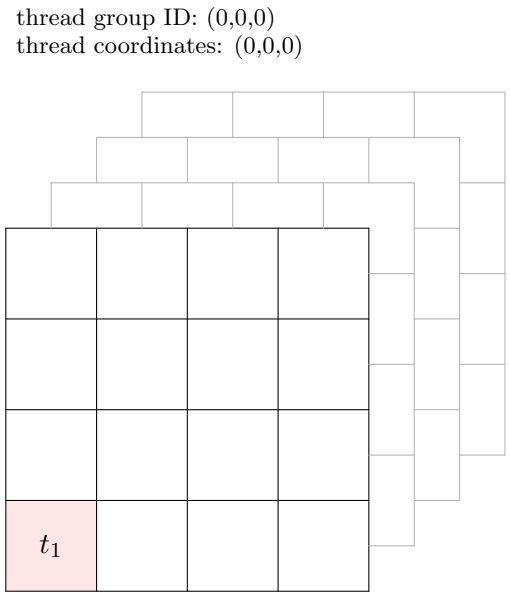


Figure 46: Compute shader thread group with dimensions 4x4x4. Marked in red is thread t_1 with $id_1 = (0,0,0)$.

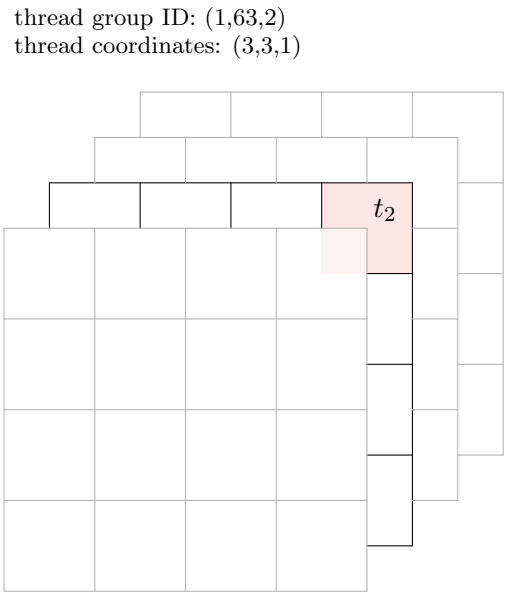


Figure 47: Compute shader thread group with dimensions 4x4x4. Marked in red is thread t_2 with $id_2 = (7,255,9)$.

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.

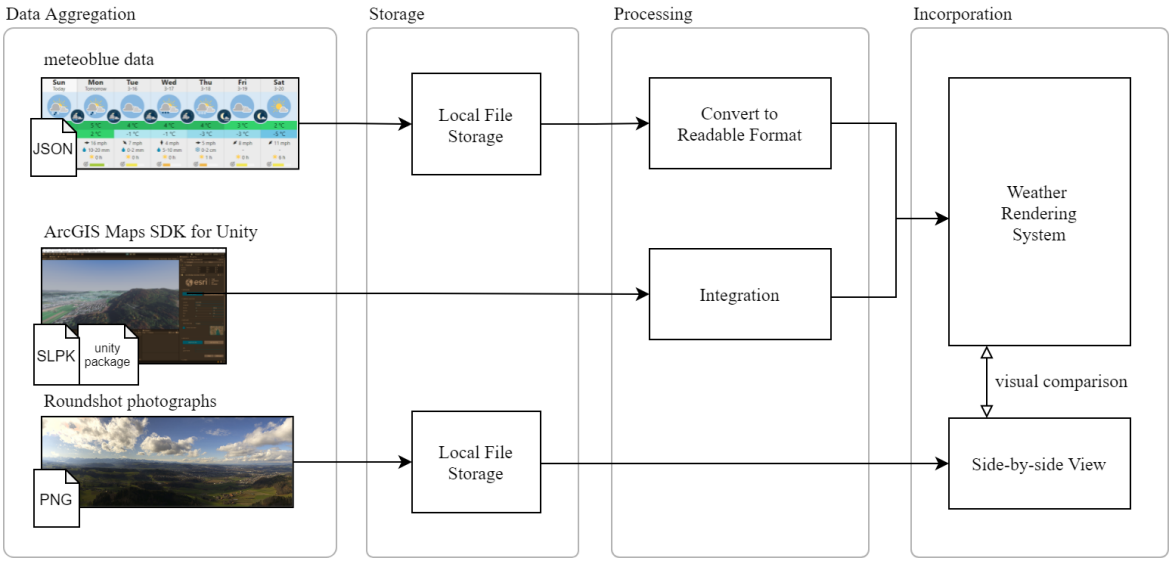


Figure 48: The system overview diagram.

5.2 Meteoblue Integration

While the bachelor project specifictation 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.

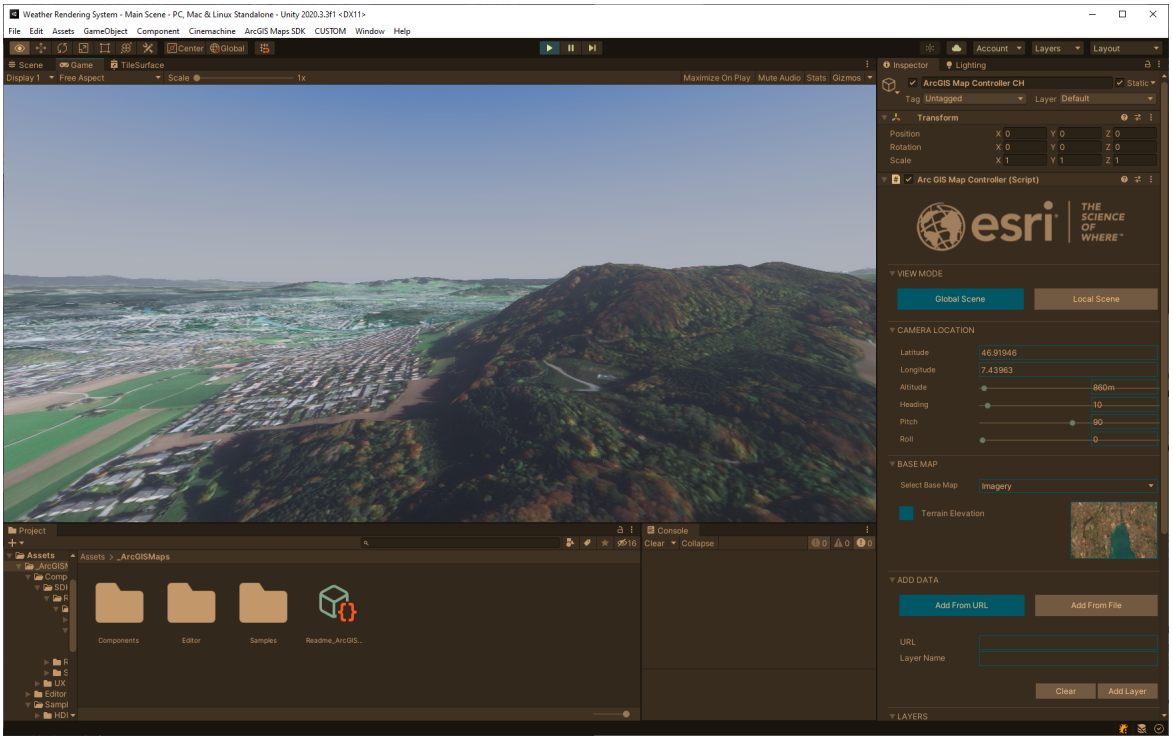


Figure 49: ArcGIS Maps SDK for Unity [24].

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.

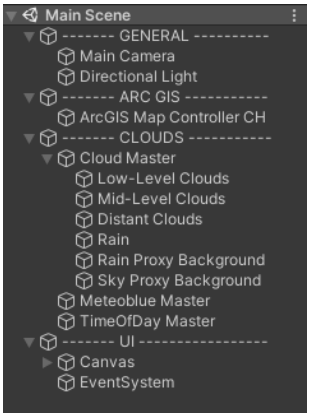


Figure 50: Hierarchy of the Unity project.

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.

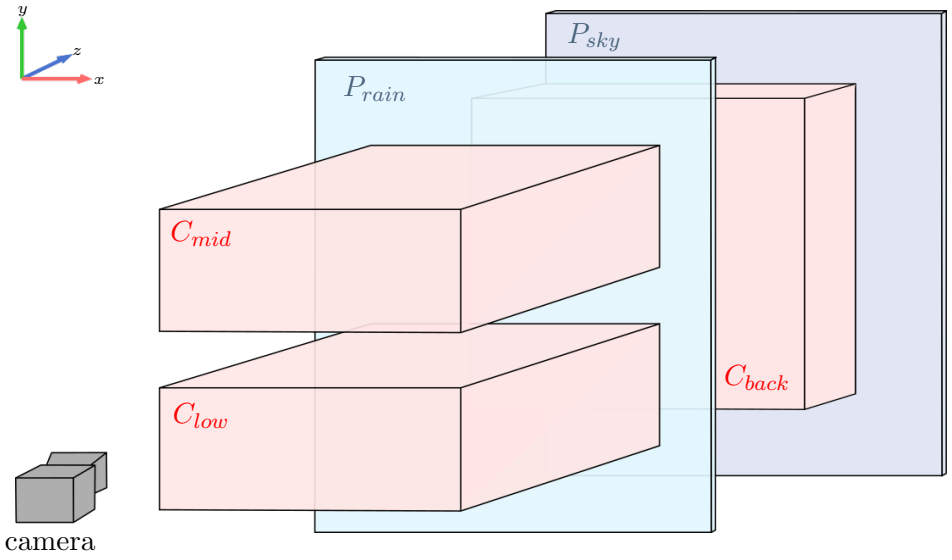


Figure 51: Final Unity scene anatomy.

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 inbetween 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.

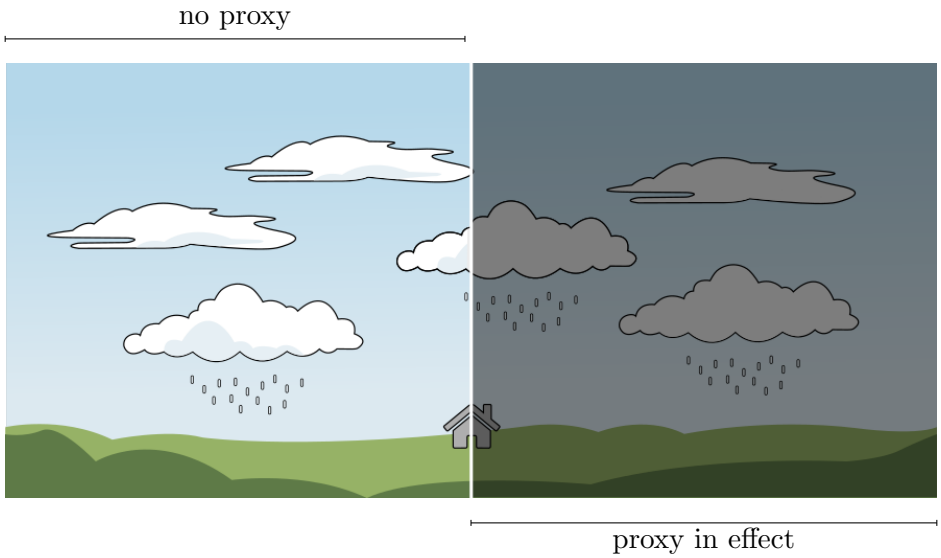


Figure 52: Desired effect of the rain proxy plane.

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.

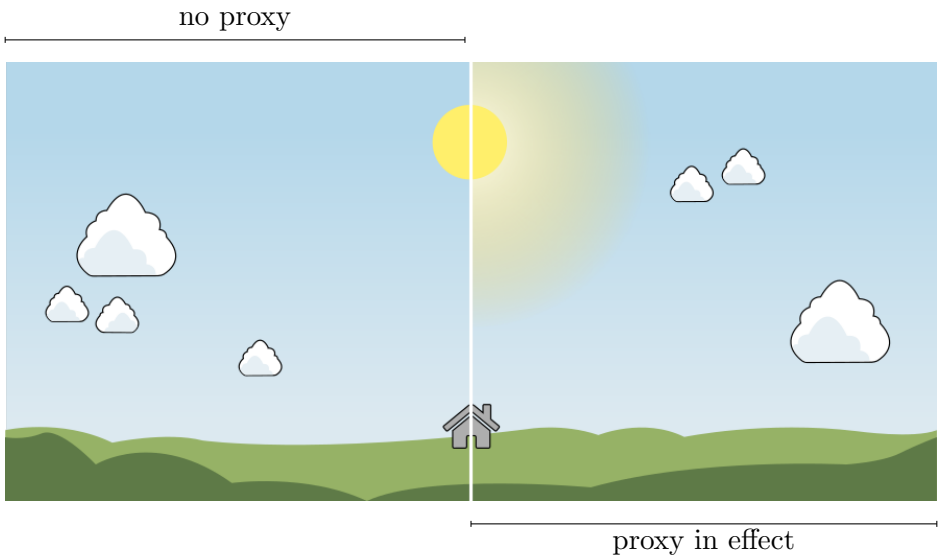


Figure 53: Desired effect of the sky proxy plane.

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).

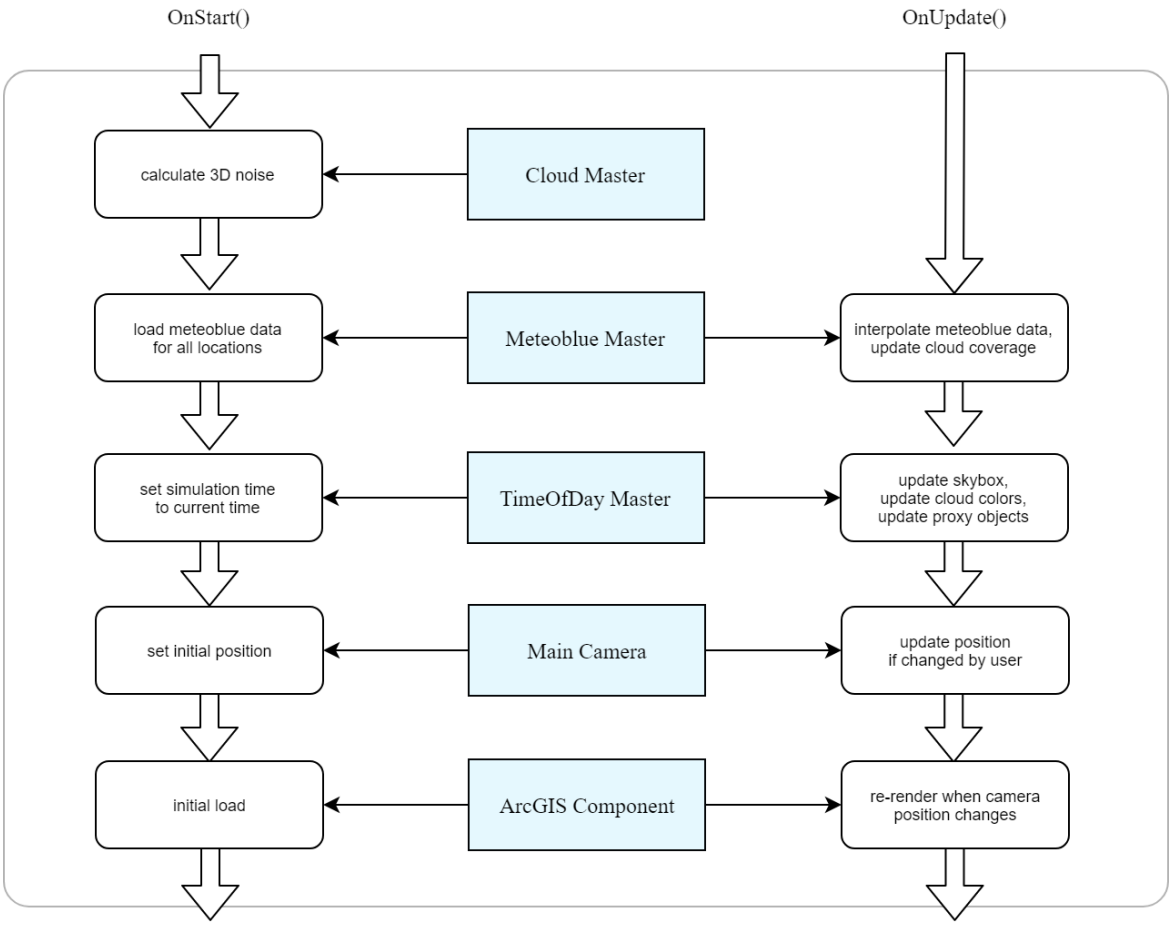


Figure 54: Render process of the Unity project implementation.

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.

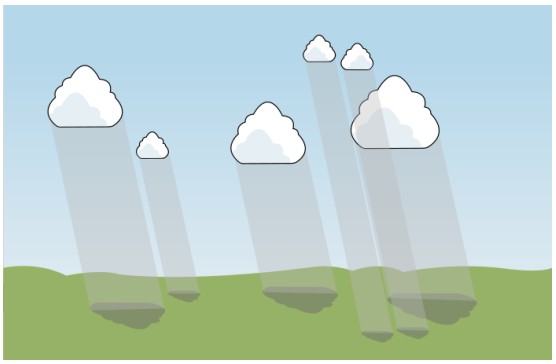


Figure 55: Shadow casting: Normally, objects would cast their own shadow in a dedicated shadow pass.

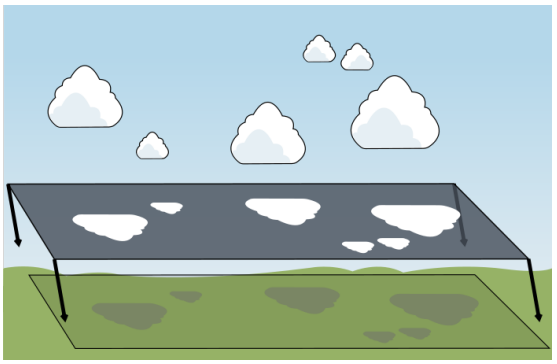


Figure 56: Shadow casting substitution: The 3D noise texture is sampled again in the ground shader and added as shadow.

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.

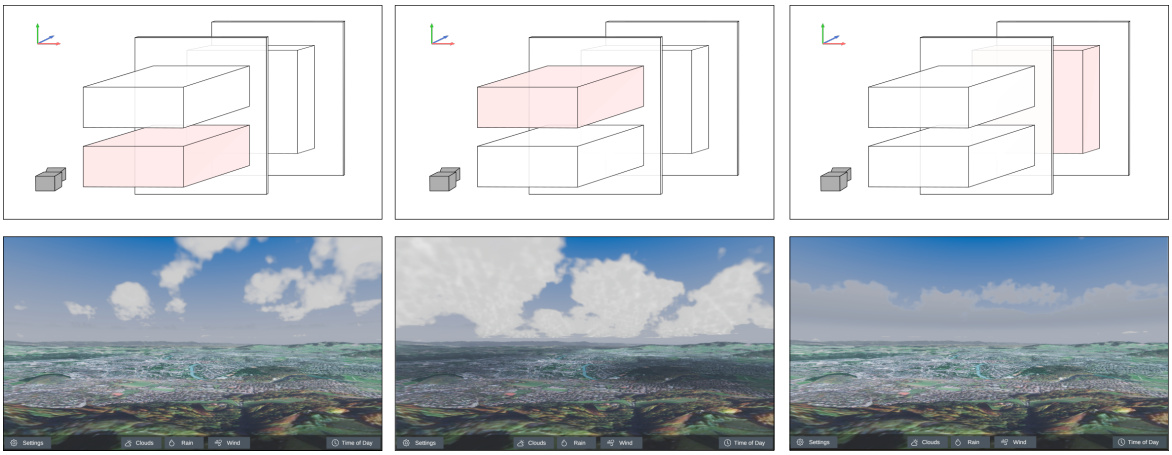


Figure 57: Side-by-side comparison of the cloud layers and their visual outputs.

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.

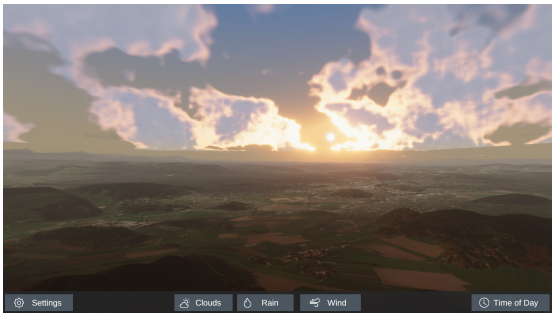


Figure 62: Final render output for a golden sunset.

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.

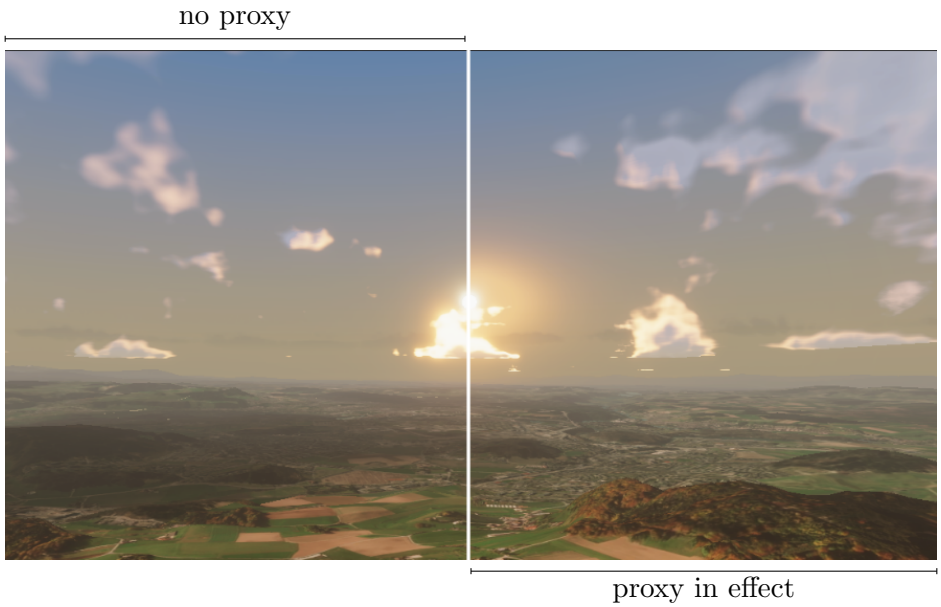


Figure 66: Visual comparison of a rendered scene with and without the sky proxy plane.

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 .

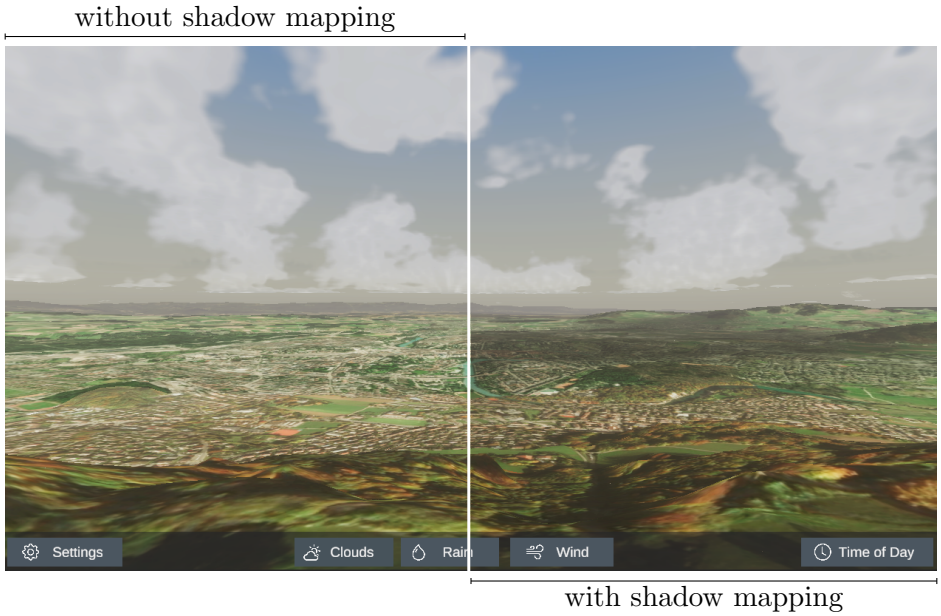


Figure 67: Visual comparison of a rendered scene with and without the shadow mapping on the ground tiles.

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 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 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 invovling 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 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 convolutional neural network was trained.

5.11.2 Generative Adversarial Network

A similar approach to the convolutional neural network is a generative adverserial network 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 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 resourceful. 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 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.

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
Fulfilled?	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
Fulfilled?	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
Fulfilled?	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
Fulfilled?	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"
Fulfilled?	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
Fulfilled?	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
Fulfilled?	Partially

6.3.2 Similar Products

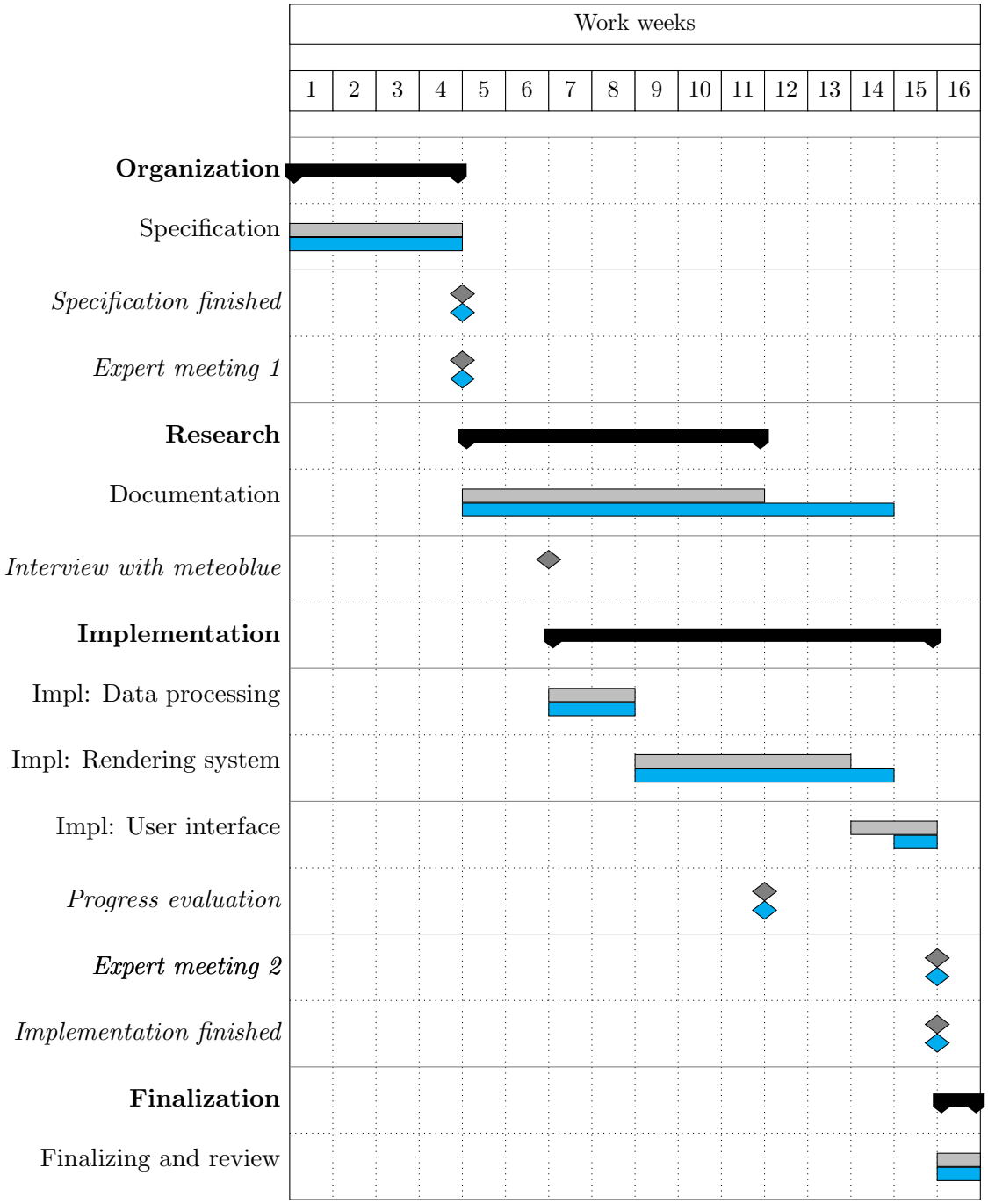
Case	T.8
Test case	Similar products
Expected result	The visual output of the weather rendering system is to be compared with the in-game footage of Microsoft's *Flight Simulator* game. The rendering system should achieve similar results, to a reasonable extent.
Actual result	✓ The visual output resembles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

7 Conclusion and Critical Discussion

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 turned out to require more time and effort.

8.2 Fulfillment of Requirements

The following table shows the original requirements and wether 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 arial images.
Since the generated 3D models from the plugin were more than a suitable replacement for the *swisstopo* data, the requirement D.3 was annuled, 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

There have been several attempts at 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 meteo data. A potential follow-up project could be an implementation of a live data feed system that continuously updated the weather rendering system.

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 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 remainder of the schedule.
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.
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.

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

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, 52, 55

Compute shader A shader which runs on the GPU but outside of the default render pipeline. 26, 27, 29, 36, 42, 51, 57

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

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

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). 42, 44

Fractal Brownian motion Different iterations of continuously more detailed noise layered on top of each other. i, 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. 50

Fragment shader A shader that processes single pixels, called fragments, calculates its color and outputs that to the frame buffer. 29, 50

Frame rate The rate at which a new image (called frame) appears on the display. 48

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, 50

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. 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, 52

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, 42, 48

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, 50

Noise A randomly generated pattern, referring to procedural pattern generation. i, 19, 20, 21, 22, 23, 24, 25, 29, 35, 36, 38, 40, 42, 55, 56, 57

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). 19, 22, 36, 48

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, 51, 55

Occlusion In meteorology, the clash of a warm front and a cold front. See occluded front. 4, 55

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, 55

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, 50, 51, 52, 55

Procedural Created solely with algorithms and independant of any prerequisites. i, 51

Proxy plane A proxy object that is a plane. 34, 39, 56

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, 51

Pseudo-random A random number generated with a deterministic algorithm, meaning that the same input will always give the same output. 19, 20, 55

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, 42, 48, 51

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, 50, 52

Shadow pass A second shader pass that only calculates the shadow of its object. 36, 56

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

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

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, 55, 56

Thermal In relation with meteorology, the hot, rising air from convection is called "thermal". 2, 3, 4, 51, 55

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

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. i, 32, 33, 36

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, 50, 55

Water vapor Evaporated water in a gaseous form. 6, 51

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, 50, 52

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`.

List of Figures

Listings