ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Čáp** Jméno: **Martin** Osobní číslo: **425456**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Simulace orografické oblačnosti

Název diplomové práce anglicky:

Simulation of orographic clouds

Pokyny pro vypracování:

Proveďte rešerši existujících metod simulace vzniku mraků používaných v počítačové grafice [1] a porovnejte jejich schopnosti generovat různé tvary mraků. Seznamte se detailně s metodou popsanou v práci [2], zaměřte se zejména na orografickou oblačnost a tuto metodu naimplementujte na GPU. Pokuste se metodu rozšířit o proudové pole větru, které bude brát v potaz tvar terénu a bude počítáno pomocí metody LBM [3-5].
Funkčnost implementované metody ověřte alespoň na třech různých scénách a výsledky porovnejte s reálnými fotografiemi orografických mraků.
Implementaci proveďte v C/C++ s využitím OpenGL / CUDA.

Seznam doporučené literatury:

[1] Y.Dobashi, K.Iwasaki, Y.Yue, T.Nishita: Visual simulation of clouds. Visual Informatics, 1(1), pp.1-8, 2017.
[2] R.P.M.A. Duarte: Realistic Simulation and Animation of Clouds using SkewT/LogP Diagrams. PhD thesis, Universidade da Beira Interior, Portugal, 2016.
[3] O. Navarro-Hinojosa, S. Ruiz-Loza, M. Alencastre-Miranda: Physically based visual simulation of the Lattice Boltzmann method on the GPU: a survey. The Journal of Supercomputing, pp.1-27, 2018.
[4] M.A. Woodgate, G.N. Barakos, R. Steijl, G.J. Pringle: Parallel performance for a real time Lattice Boltzmann code. Computers & Fluids, 2018.
[5] M.F. King, A. Khan, N. Delbosc, H.L. Gough, C. Halios, J.F. Barlow, C.J. Noakes: Modelling urban airflow and natural ventilation using a GPU-based lattice-Boltzmann method. Building and Environment, 125, 2017, pp.273-284.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jaroslav Sloup, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.02.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **20.09.2020**

Ing. Jaroslav Sloup
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's Thesis

Simulation of Orographic Clouds

Bc. Martin Čáp

Supervisor: Ing. Jaroslav Sloup

Study Programme: Open Informatics

Specialization: Computer Graphics

May 22, 2019

Acknowledgement

I would like to express my gratitude to my supervisor Ing. Jaroslav Sloup for his constant guidance and valuable advice during creation of this thesis. Furthermore, I would like to thank my family and friends for their continuous support during my studies.

Declaration

I declare that I have created the presented thesis independently and that I have quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic theses.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Prague, May 22, 2019 ....

Abstract

Realistic cloud simulation is a hot topic in computer graphics research. In this thesis we present a possible approach of modeling orographic clouds using SkewT/LogP diagrams as proposed by Rui Pedro Duarte. We extend this simulation with well-known Lattice Boltzmann method to simulate wind and its effects on clouds in mountainous environments. We parallelize both methods on the GPU using CUDA technology by NVIDIA. Lastly, we propose a simple extension of volumetric rendering method by Simon Green that uses phase functions to enhance graphical fidelity of our cloud images. We present this simulator in a simple to use application that gives its users a wide variety of tools to create desired results.

Keywords: cloud dynamics, particle systems, SkewT/LogP diagrams, Lattice Boltzmann method, CUDA

Abstrakt

Realistická simulace mraků je žhavým tématem výzkumu v oboru počítačové grafiky. V této práci představíme možný způsob modelování orografických mraků za pomoci SkewT/LogP diagramů, jež navrhl Rui Pedro Duarte ve své disertační práci. Simulaci mraků jsme rozšířili o známou Lattice Boltzmannovu metodu pro simulaci větru a jeho vlivu na mraky v hornatých prostředích. Obě metody paralelizujeme na GPU za použití CUDA technologie od společnosti NVIDIA. Na závěr představíme jednoduché rozšíření Simon Greenovy metody pro vykreslování volumetrických dat, které používá fázovou funkci pro realističtější zobrazení mraků. Simulátor demonstrujeme v aplikaci, jejíž použití je jednoduché a nabízí velké množství nástrojů pro tvorbu kýžených výsledků.

Klíčová slova: dynamika mraků, částicové systémy, SkewT/LogP diagramy, Lattice Boltzmannova metoda, CUDA

Contents

1 Introduction **1**
1.1 Cloud Classification . . . 2
1.2 Orographic Clouds . . . 2

2 Cloud Dynamics **5**
2.1 Parcel Theory . . . 5
2.2 Equations of Cloud Motion . . . 5

3 Related Works **9**
3.1 Physically-Based Methods . . . 10
3.2 Procedural Methods . . . 13
  3.2.1 Cellular Automata . . . 13
  3.2.2 Particle Systems . . . 14
  3.2.3 Volumetric Techniques . . . 17
  3.2.4 Image-Based Methods . . . 19
3.3 Closing Remarks . . . 19

4 Cloud Simulation Using SkewT/LogP Diagrams **21**
4.1 SkewT/LogP Diagram . . . 21
  4.1.1 Main Curves . . . 22
  4.1.2 Sounding Curves . . . 24
4.2 Simulation Steps . . . 25
  4.2.1 Birth . . . 26
  4.2.2 Dry Lift and Moist Lift . . . 27
4.3 Orographic Clouds . . . 29
  4.3.1 Terrain Influence . . . 31

5 Lattice Boltzmann Method **33**
5.1 Streaming Step . . . 34
5.2 Collision Step . . . 35
5.3 Inlets . . . 38
5.4 Obstacles and Boundaries . . . 38
5.5 Particle Advection . . . 38
5.6 Main Loop . . . 39

6 Cloud Rendering 41
6.1 Algorithm . . . 41
6.2 Particle Rendering . . . 42
6.3 Cast Shadows . . . 43
6.4 Light Texture Blurring . . . 43
6.5 Cloud Occlusion . . . 44
6.6 Phase Function . . . 44
  6.6.1 Isotropic Phase Function . . . 45
  6.6.2 Rayleigh Phase Function . . . 45
  6.6.3 Henyey-Greenstein Phase Function . . . 46
  6.6.4 Double Henyey-Greenstein Phase Function . . . 47
  6.6.5 Schlick Phase Function . . . 47
  6.6.6 Cornette-Shanks Phase Function . . . 47

7 Solution Proposal 49
7.1 SkewT/LogP Simulation . . . 50
7.2 Lattice Boltzmann Method Simulation . . . 50
7.3 Viewports . . . 51
7.4 Terrain . . . 52
7.5 Cloud Rendering and Sky . . . 52
7.6 Rendering . . . 52
7.7 Debugging Tools and User Control . . . 52
7.8 World Unit Size . . . 53

8 Implementation 55
8.1 Particle System and Memory Management . . . 55
8.2 SkewT/LogP Diagrams . . . 56
  8.2.1 Dry Adiabat Creation . . . 57
  8.2.2 Moist Adiabats Implementation Details . . . 57
  8.2.3 Intersection of Line Segments in 2D . . . 59
  8.2.4 Diagram Customization . . . 62
  8.2.5 Text Rendering . . . 63
8.3 SkewT/LogP Simulator . . . 63
8.4 Lattice Boltzmann Method . . . 65
  8.4.1 Standalone Application . . . 67
  8.4.2 Inlets . . . 67
  8.4.3 Boundaries . . . 67
  8.4.4 Streamline Particle System . . . 68
8.5 Cloud Rendering . . . 69
  8.5.1 Particle Sorting . . . 69
  8.5.2 Slice Rendering . . . 71
  8.5.3 Auxiliary Framebuffers . . . 72
  8.5.4 Phase Function Integration . . . 73
  8.5.5 Performance Note . . . 73

8.5.6 Stylization 74
8.6 Sky Rendering 75
8.7 Terrain 77
8.7.1 Heightmap 77
8.7.2 Perlin Noise 78
8.7.3 Creation 79
8.7.4 Texturing 80
8.8 Emitters 81
8.8.1 Cumulative Distribution Function Sampling 81
8.8.2 Brush Mode 82
8.8.3 Memory Management 84
8.9 General 85

9 Results 87
9.1 Measurements 92
9.1.1 CPU and GPU Comparison 94

10 Conclusion 97

11 Future Work 99

Bibliography 101

A Obtaining Relation Between T and P 109

B Exponential Variance Shadow Maps 111

C Subgrid Model of LBM 113

D Physics and Thermodynamics Basics 115

E Point Sprite Rendering 119

F User Interface 121

G Acronyms 133

H DVD Contents 135

List of Figures

1.1 Cloud classification . . . 2
1.2 Orographic lifting on the windward side . . . 3
1.3 Orographic influence on the leeward side . . . 4
1.4 Examples of common orographic clouds . . . 4

2.1 Parcel theory . . . 6

3.1 Scene from the popular TV show Game of Thrones . . . 9
3.2 Examples of clouds in recent video game releases . . . 10
3.3 Taxonomy of cloud simulation methods . . . 11
3.4 Physically-based methods . . . 12
3.5 Procedural methods that use particle systems . . . 15
3.6 Procedural methods that use particle systems (*continued*) . . . 16
3.7 Volumetric procedural methods . . . 18
3.8 Image-based methods . . . 20

4.1 Simulation stages of Duarte's method . . . 21
4.2 Comparison of implemented SkewT/LogP diagram with its source . . . 25
4.3 SkewT/LogP diagram with highlighted curves . . . 26
4.4 Trajectories of particles for dry and moist lift . . . 28
4.5 Diagram showing orographic parameters . . . 30
4.6 Obtaining $\vec{v}_{new}$ for terrain influence . . . 31

5.1 Visualization of distribution function f_i . . . 33
5.2 Streaming step for D2Q9 configuration . . . 34
5.3 Selected DdQq orderings for our implementation . . . 35
5.4 Flow around circular obstacle with different τ values . . . 37
5.5 Full bounce back model . . . 39
5.6 LBM simulation step order . . . 39

6.1 Single and multiple scattering of light inside a cloud . . . 41
6.2 Half-angle vector computation . . . 42
6.3 Comparison of cloud visualization with and without blur . . . 44
6.4 Phase function scattering angle ϕ . . . 45
6.5 Plots of Rayleigh, Henyey-Greenstein and Mie phase functions . . . 46
6.6 Henyey-Greenstein phase function with different g values . . . 48
6.7 Comparison of different phase functions . . . 48

7.1 Proposed architecture of the application 49
7.2 Proposed viewport of our application 51

8.1 Moist adiabats obtained by using non-iterative methods 58
8.2 Example of a problematic diagram 64
8.3 Isobar intersection clamping 64
8.4 Box displaced by LBM 66
8.5 2D visualization of particle velocities using viridis color map 67
8.6 Streamlines generated around a peak 68
8.7 Rendering loop of the application 73
8.8 Comparison of clouds with different particle counts 74
8.9 Stylized cloud appearance using a texture atlas 74
8.10 Angles used in sky model computation 75
8.11 Sunrise generated with the Hošek-Wilkie model 76
8.12 Heightmap and its corresponding terrain 77
8.13 Example of terrains generated by perlin noise with 10 octaves 78
8.14 Terrain texture visualizations 80
8.15 Class diagram of implemented emitters 81
8.16 Cloud formation with a custom shape 82
8.17 Clouds generated using perlin noise 83
8.18 Hand drawn cloud shape 84
8.19 Main loop of the application 85

9.1 Fog-like clouds flowing through a mountain range 87
9.2 Clouds generated with perlin noise emitters 88
9.3 Thick cloud layer 88
9.4 Mountain peak cloud comparison 89
9.5 Cloud rendering comparison 89
9.6 Cloud layer comparison 90
9.7 Clouds flowing above a mountain range 90
9.8 Large cloud generated by our method 91
9.9 Stratocumulus cloud comparison 91
9.10 Benchmark of our application with 500 thousand particles 93
9.11 Screenshot obtained during benchmarking 94
9.12 Benchmark of our application with 10 million particles 95

B.1 Example of light bleeding 111
B.2 Shadow mapping techniques comparison 112

F.1 User interface overview 122
F.2 LBM tab 123
F.3 SkewT/LogP tab 124
F.4 Cloud visualization tab 125
F.5 Lighting tab 126

F.6 Terrain tab 127
F.7 Sky tab 128
F.8 Emitters tab 129
F.9 View tab 130
F.10 Particles tab 130
F.11 Debug tab 131
F.12 Hierarchy tab 132
F.13 Properties tab 132

Listings

8.1 Mapping functions for diagram creation 56
8.2 Dry adiabat creation 57
8.3 Moist adiabat creation 59
8.4 Isobar intersection using binary search 62
8.5 Particle distances kernel 70
8.6 Particle sorting algorithm 70
8.7 Multiple octave perlin noise generator 78
8.8 Normal vector calculation from height data 80

1 Introduction

Clouds play a large role in visualizing expressive outdoor scenes in multiple media such as movies and video games. Shapes, movement and appearance of clouds have been studied for decades in the field of computer graphics. Their research spawned countless methods ranging from realistic fluid simulations to purely procedural approaches. Our main focus in this thesis are orographic clouds which are formed in mountainous environments. To achieve this, we utilize so-called SkewT/LogP (STLP) method for simulating cloud dynamics as proposed by Duarte in his dissertation thesis [Dua16]. We present a parallel implementation of the STLP method with two improvements. First, Duarte's wind simulation uses sounding data to naively advect particles, whereas our implementation uses a flow field generated by Lattice Boltzmann method (LBM) which is a staple in the field of computational fluid dynamics (CFD). Secondly, we parallelize both STLP and LBM on GPU using NVIDIA CUDA technology to achieve real-time results on common hardware for large amounts of particles.

To evaluate our generated cloud shapes, we have implemented a volumetric data rendering technique presented by Green [Gre08]. Since this method was used primarily to simulate smoke, we have extended it with anisotropic light scattering by using phase functions to more closely resemble clouds. This is particularly apparent when the observer is looking directly at the sun where the silhouette of the observed cloud is much more pronounced due to light passing through its low density regions.

We incorporate all these methods into our framework that was implemented in C++ and OpenGL/CUDA. The framework provides a vast range of options and parameters to customize the simulation. Furthermore, additional tools such as terrain generation and custom shaped emitters are present to make the process of cloud creation and simulation as easy to use as possible.

In the rest of this chapter, the general cloud classification and a short description of orographic clouds will be presented. Relevant cloud dynamics theory that is used by Duarte [Dua16] will be covered in Chapter 2. In Chapter 3, selected important methods from the field of cloud simulation and rendering will be examined. In Chapter 4, Chapter 5 and Chapter 6 we will look at each of the main three methods used in this thesis: STLP, LBM and cloud rendering, individually. In Chapter 7 we propose an integration of these methods into a single framework and in Chapter 8 the implementation details are presented. In Chapter 9 we present results obtained with our method and performance measurements. Lastly, in Chapter 10 we conclude our work and in Chapter 11 we discuss possible future improvements.

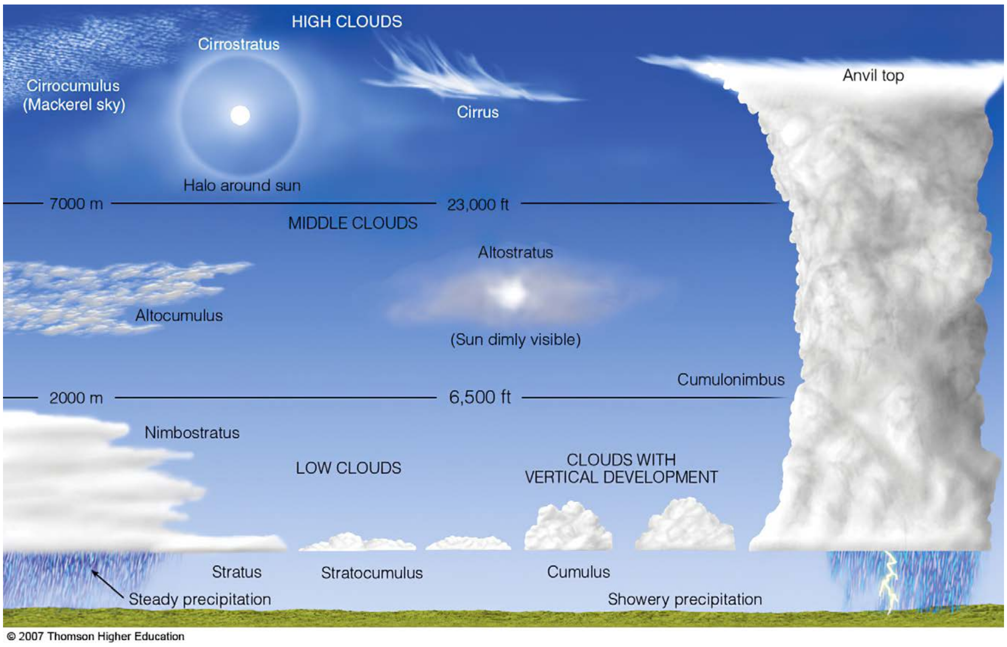

HIGH CLOUDS
Cirrostratus
Cirrocumulus
(Mackerel sky)
Cirrus
Anvil top
Halo around sun
7000 m
23,000 ft
MIDDLE CLOUDS
Altostratus
Altocumulus
(Sun dimly visible)
Cumulonimbus
2000 m
6,500 ft
Nimbostratus
LOW CLOUDS
CLOUDS WITH
VERTICAL DEVELOPMENT
Stratus
Stratocumulus
Cumulus
Steady precipitation
Showery precipitation
© 2007 Thomson Higher Education

Figure 1.1: Cloud classification diagram [Ahr11].

1.1 Cloud Classification

Clouds are a large group of tiny water droplets or ice crystals that are suspended in the air. They are aesthetically pleasing elements of our atmosphere that come in many shapes and forms. Generally, clouds are categorized into four major groups based on their altitude, appearance and characteristics. These are low clouds, middle clouds, high clouds, and clouds with vertical development [Ahr11]. This classification has its origins in 1803 when it was proposed by an English naturalist Luke Howard and has been widely accepted since. Howard's system employed Latin words to describe clouds as they appear to the observer. As an example, cumulus clouds, which are the main focus of Duarte's thesis, can be translated as "heap" clouds due to their puffy shape. In 1887, Ralph Abercromby and Hugo Hildebrandsson expanded Howard's system by separating the clouds into the mentioned four major groups while keeping Howard's system to classify individual subgroups within them as shown in Figure 1.1.

1.2 Orographic Clouds

There are four major mechanisms that lead to cloud development: surface heating followed by free convection, orographic lift, ascent due to convergence of surface air, and uplift along weather fronts [Ahr11]. Out of these four we are mainly interested in the first two. Surface heating and free convection will be examined in the cloud dynamics oriented Chapter 2. Let us now look at the main topic of this thesis, orographic clouds.

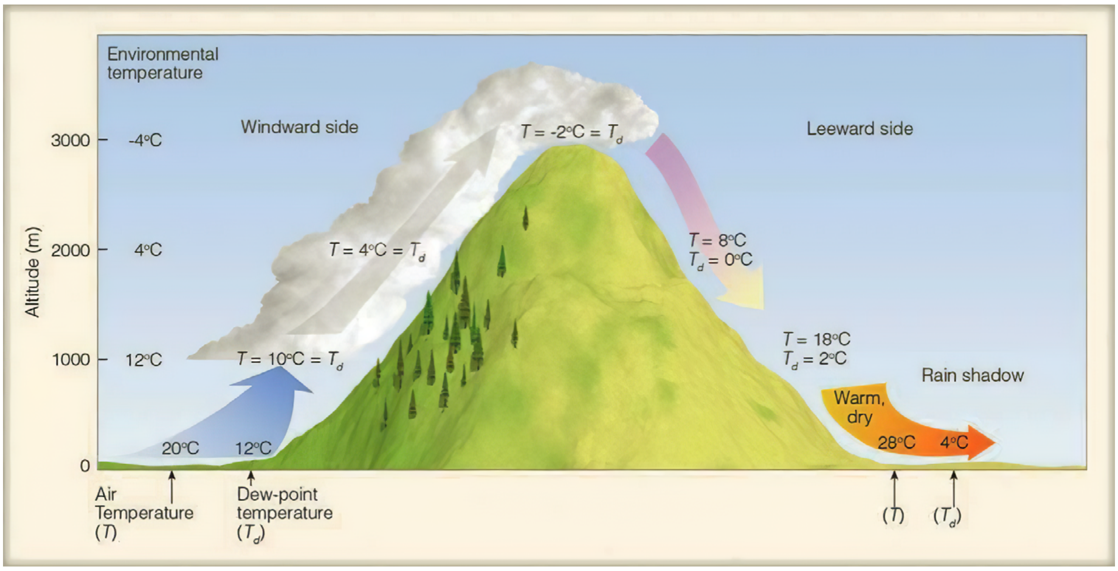
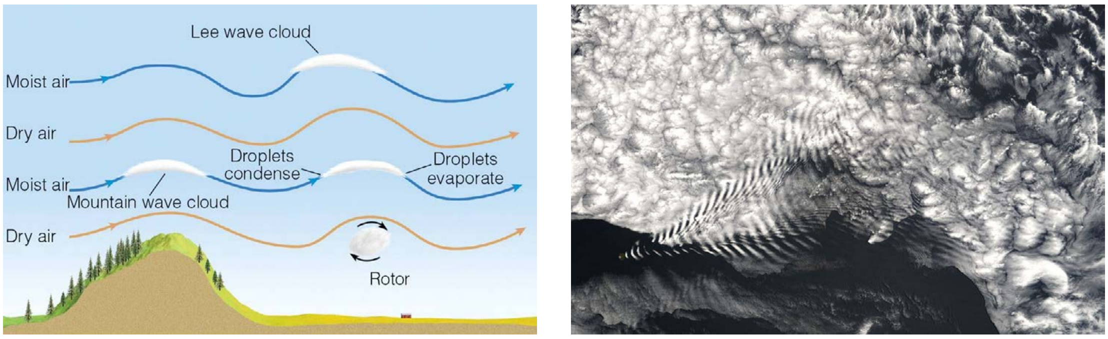

Environmental temperature
Altitude (m)
3000
2000
1000
0
-4°C
4°C
12°C
Windward side
T = -2°C = T_d
Leeward side
T = 4°C = T_d
T = 8°C
T_d = 0°C
T = 10°C = T_d
T = 18°C
T_d = 2°C
Rain shadow
Warm, dry
20°C
12°C
28°C
4°C
Air Temperature (T)
Dew-point temperature (T_d)
(T)
(T_d)

Figure 1.2: Diagram showing orographic lifting for moist air on the windward side [Ahr11].

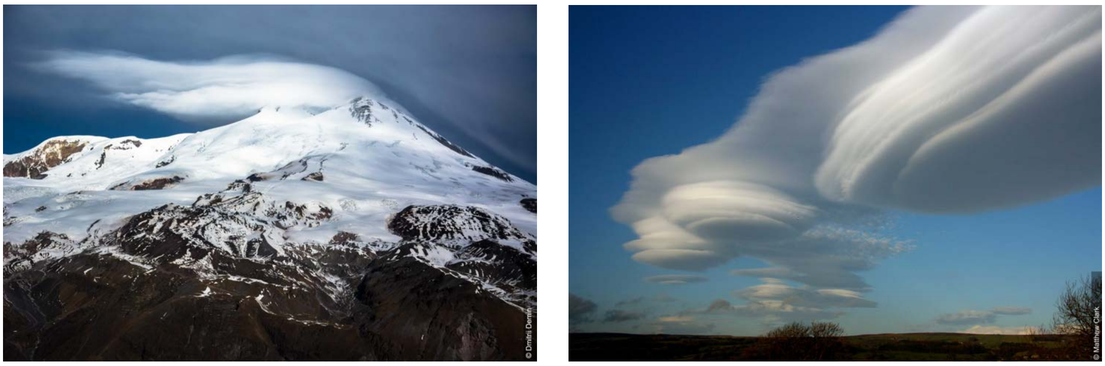
Orographic clouds can belong to either major cloud group depending on their cloud base height [Py16]. They are produced by the second mentioned mechanism of cloud development: orographic lift. Orographic lift occurs when an airflow encounters a mountain or another type of an obstacle, and is forced to rise. If the flow is sufficiently humid, clouds form on the windward side of mountains as depicted in Figure 1.2. Orographic influence is not limited to the windward side of the mountain. On the leeward side, particularly in areas with strong winds, the airstream that was disturbed by a barrier starts to oscillate as it moves downstream, generating mountain waves [wmob]. These conditions produce so-called wave clouds that are usually formed from lenticular clouds. Example of wave clouds that were captured from a satellite along with diagram showing the process of their creation is shown in Figure 1.3. Another cloud types that are formed through orographic influence can be seen in Figure 1.4.


Lee wave cloud
Moist air
Dry air
Droplets condense
Droplets evaporate
Moist air
Mountain wave cloud
Dry air
Rotor

(a) Diagram showing orographic influence on the leeward side that produces wave clouds [Ahr11].

(b) Satellite image of wave clouds composed of lenticular clouds flowing around volcanic Amsterdam Island [Sch].

Figure 1.3: Orographic influence on the leeward side creates wave clouds with streamline pattern that can be observed from high altitudes.


© Dmitrii Demin
© Matthew Clark

(a) Banner cloud [wmoc].

(b) Lenticular clouds [wmod].

Figure 1.4: Examples of common cloud types that are formed through orographic influence.

2 Cloud Dynamics

Realistic visualization of clouds can be decomposed into multiple smaller parts. These would fall under two major groups, dynamics and radiometry as described by Harris in his dissertation thesis [Har03]. Cloud dynamics describe the motion of air and its effects on cloud formation. Dynamics are also concerned with meteorological processes such as phase changes, mainly condensation and evaporation of water. Cloud radiometry is the study of how light interacts with clouds [Har03]. In this thesis, our initial goal was mainly oriented towards realistic cloud dynamics. Because cloud radiometry is tackled by using a procedural method, radiometry is only briefly mentioned in Chapter 6 where anisotropic light scattering is described.

2.1 Parcel Theory

Let us say that an *air parcel* is an observed small mass of air that has slightly different characteristics than the air that surrounds it, usually called environmental air. The air parcel is influenced by the environment, but it does not affect the environment itself. The parcel has a different temperature, composition and density than the surrounding air and thanks to this moves through the environment as a result [And10]. Basic principle is shown in Figure 2.1 where we can see a parcel of air expand and cool as it moves upwards and vice-versa when descending. This is initiated when ground is warmed enough by the sun for a parcel to start rising which is the first mechanism of cloud formation as mentioned previously.

2.2 Equations of Cloud Motion

Let us look at theory behind the convection process (ascension and descension) of cumulus clouds as shown in Figure 2.1. The upcoming section is paraphrased and shortened from Duarte's thesis [Dua16].

Majority of the Earth's atmosphere up to an altitude of about 90km is composed of ideal gases such as nitrogen (78%), oxygen (21%) and a variety of trace gases. All these gases obey the ideal gas law [Har03]. The law states that each of its moles follows the equation

$$PV_m = RT \tag{2.1}$$

where P is the pressure, V_m is the volume of one mole, R is the universal gas constant and T is the absolute temperature. The corresponding law for unit mass of air by denoting M_m as mass of one mole is then described as

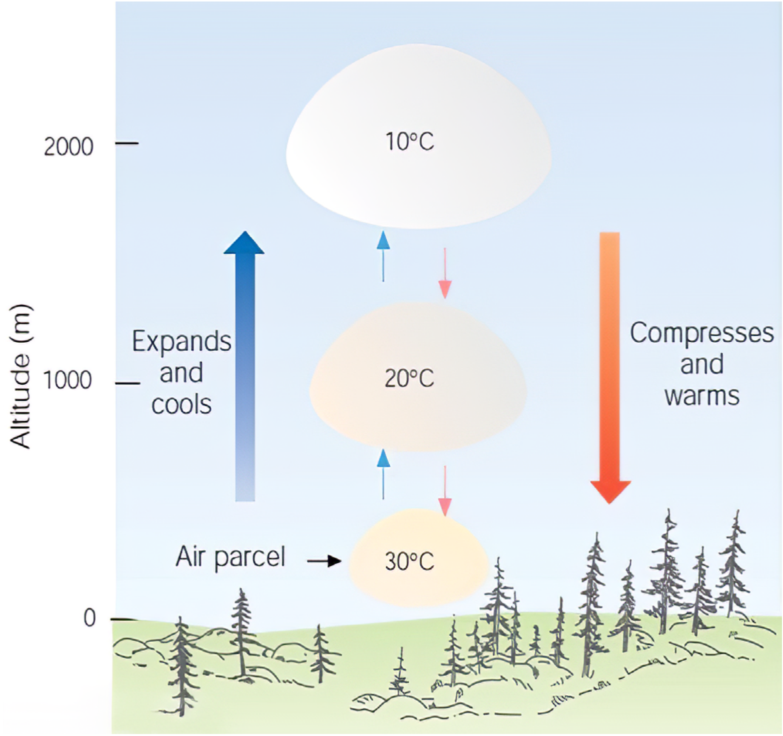


Figure 2.1: Basic idea of parcel theory, surface heating and convection [Ish].

$$P = R_a T \rho \tag{2.2}$$

where $R_a = R/M_m$ [$\mathrm{Jkg}^{-1}\mathrm{K}^{-1}$] is the gas constant per unit mass of air. The gas constant R_a takes value $R_d = 287.05307$ for dry air and $R_m = 461.5$ for moist air (sometimes also denoted as specific gas constant for water vapor R_v) [And10, Dua16].

At the start, when air has no liquid water, it is called *dry air*. A dry air parcel is described by three properties: absolute temperature T in K, pressure P in $\mathrm{Pa} = \mathrm{N/m}^2$ and density ρ in $\mathrm{kg/m}^3$. As the air expands in response to the decrease in pressure, its temperature also decreases, resulting in a process called *adiabatic expansion*. Generally, adiabatic processes are such processes where no heat is transferred between the air parcel and its surrounding environment.

Gravity is the most important external force acting on the atmosphere. The environment pressure P_e and parcel pressure are said to be in *hydrostatic balance* when

$$\left(\frac{dP_e}{dz}\right)_z \approx -\rho_e(z)g \approx -\rho_p(z)g \tag{2.3}$$

where ρ_e is the environment density, ρ_p is the parcel density, g is the acceleration due to gravity, and z is the air parcel altitude. Left side of Equation 2.3 ($(dP_e/dz)_z$) is the upward force (environmental pressure gradient) and $-\rho_p(z)g$ is the downward force. According to Newton's law of motion, we have

$$\rho_p(z)\frac{dv}{dt} = -\left(\frac{dP_e}{dz}\right)_z - \rho_p(z)g \tag{2.4}$$

where dv/dt is the vertical acceleration of the air parcel. Once the air parcel has been lifted to altitude z, it has the same pressure as the environment, that is, $P_e(z) = P_p(z)$. Using the Equation 2.2 ($P = \rho R_a T$), we obtain

$$\frac{dv}{dt} = g\frac{T_p(z) - T_e(z)}{T_e(z)} \tag{2.5}$$

which describes the vertical acceleration of the parcel with dependence on its ambient temperature $T_p(z)$ and ambient temperature of the environment surrounding it $T_e(z)$. To see how we got to this equation, please see Appendix A.

Relation between the temperature and pressure of a gas under adiabatic changes is defined as

$$\frac{T}{T_0} = \left(\frac{P}{P_0}\right)^k \tag{2.6}$$

where T_0 and P_0 are the initial temperature and pressure, and T and P are the temperature and pressure after the adiabatic change. The exponent k is defined as

$$k = \frac{R_d}{c_{pd}} = \frac{c_{pd} - c_{vd}}{c_{pd}} \approx 0.286 \tag{2.7}$$

where c_{pd} and c_{vd} are the specific heat capacity of dry air at constant pressure and volume, respectively.

Potential temperature θ is a more convenient variable to account for adiabatic changes of temperature and pressure since its value is constant under adiabatic changes of altitude. It is defined as a temperature that an air parcel would have if it were moved adiabatically from pressure P and temperature T to pressure P_0. It is given by

$$\theta = T\left(\frac{P_0}{P}\right)^{R_d/c_{pd}} \tag{2.8}$$

By substituting absolute temperature with potential temperature in Equation 2.5 we obtain

$$\frac{dv}{dt} = g\frac{\theta_p(z) - \theta_a(z)}{\theta_a(z)} \tag{2.9}$$

This equation can be used to determine the vertical displacement of an air parcel by computing the potential temperature of the parcel itself and its surrounding environment. This is crucial for the method since computation of these potential temperatures is possible with utilization of a SkewT/LogP diagram that is described in Chapter 4.

3 Related Works

Before delving into the selected methods used in this thesis, let us first look at other works that pertain to cloud simulation. Clouds play an important role in a plethora of different media, most noticeably in movie and video game industries. In movies and television, clouds may be integral to the story or to the visual setting. A nice recent example is the scene shown in Figure 3.1 from the popular TV series Game of Thrones.

Figure 3.1: Scene from the popular TV show Game of Thrones showing captivating vista composed of heavy cloud layers.

Nowadays, a lot of focus has been shifted to procedural cloud visualization for large open worlds in the video game industry with games such as Horizon Zero Dawn or Red Dead Redemption 2 whose cloudscapes are shown in Figure 3.2. Clouds and the atmosphere have become an important part of all modern game engines such as Decima Engine, Frostbite, Unreal Engine and others. In many applications, the importance of clouds is not only visual, but practical as well. These are usually flight or army simulators that are used to train professionals in their respective industries.

Because clouds are such an important aspect of visualizing outdoor scenes, many methods were proposed since the very beginning of computer graphics history. Their classification differs from author to author, but the general outline is the same for all.

(a) Horizon Zero Dawn [SV15]. (b) Red Dead Redemption 2.

Figure 3.2: Examples of clouds in recent video game releases.

There are two main categories into which we can divide all methods that are used to simulate clouds, physically-based methods and procedural methods as shown in Figure 3.3. Furthermore, Dobashi divides methods based on their main purpose into cloud modeling, animation and cloud rendering [DIYN17]. Similarly, Harris divides the methods into cloud dynamics (modeling and animation) and radiometry (rendering) as mentioned earlier [Har03]. Lines between the categories can become quite blurred as we'll see in the upcoming sections. This is due to the fact that multitude of proposed methods combine ideas from all these subcategories. Additionally, cloud dynamics and rendering are dependent in terms of data representation. For example, if a simulation uses a particle system, then the rendering process either also makes use of said particle system or the data have to be converted to different format which is usually a costly operation. With this in mind, let us look at both physically-based methods and procedural methods and list some of their important representatives.

3.1 Physically-Based Methods

Physically-based methods are predominantly used to model realistic physical behaviour of fluids and fall into the field of computational fluid dynamics (CFD). Since these methods mainly focus on realism of the simulation, they do not necessarily take visually pleasing results into account. Furthermore, physically-based methods are in many cases computationally expensive, especially for large volumes of fluids as is the case with cloud formations spanning vast terrains.

Physically-based methods involve solving the Navier-Stokes equations. Navier-Stokes equations are partial differential equations defined as follows:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \tag{3.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{3.2}$$

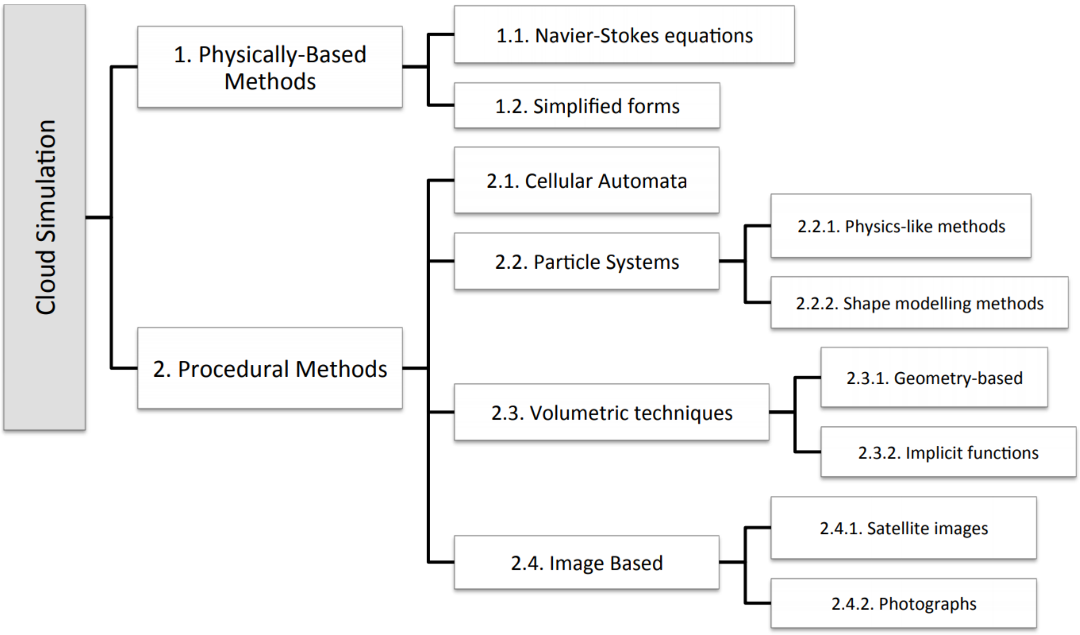
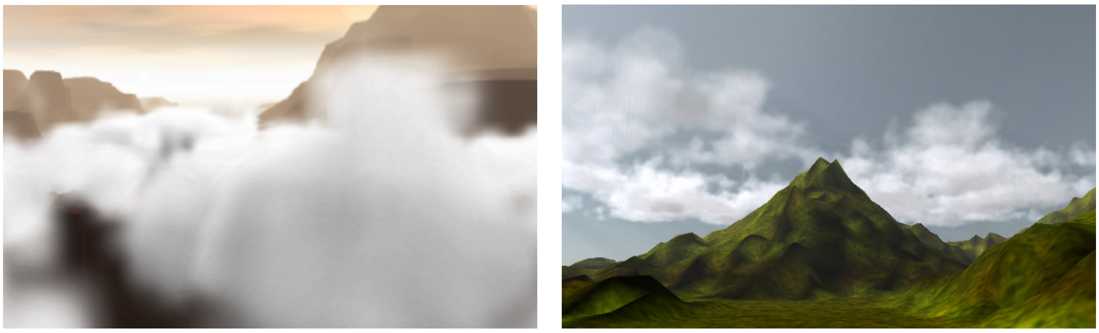

Cloud Simulation
1. Physically-Based Methods
1.1. Navier-Stokes equations
1.2. Simplified forms
2. Procedural Methods
2.1. Cellular Automata
2.2. Particle Systems
2.2.1. Physics-like methods
2.2.2. Shape modelling methods
2.3. Volumetric techniques
2.3.1. Geometry-based
2.3.2. Implicit functions
2.4. Image Based
2.4.1. Satellite images
2.4.2. Photographs

Figure 3.3: Diagram showing basic taxonomy of cloud simulation methods [Dua16].

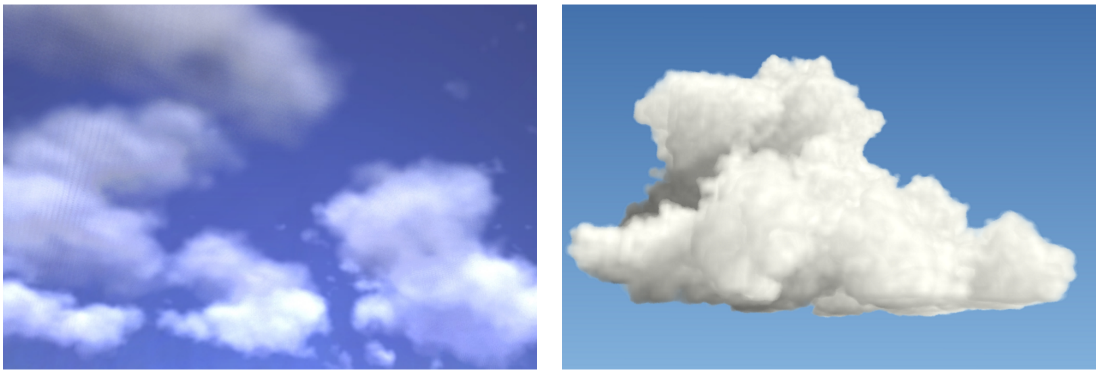
where $\vec{u}$ stands for the velocity of the fluid, ρ the density of the fluid, p the pressure, $\vec{g}$ the acceleration due to gravity and ν denotes the kinematic viscosity which measures how viscous the fluid is [Dua16]. To solve these equations, computationally expensive techniques need to be used in most cases.

Physically-based approach gained a lot of traction in the year 1999 when Jos Stam introduced a stable method to solve Navier-Stokes equations [Sta99]. This means that the simulation was guaranteed not to diverge. Since then, a lot of research was built on this method, including plethora of cloud simulations. A notable example is a discrete stable solver by Harris et al. that simulates cloud dynamics on the GPU. The solver runs at real-time speeds by utilizing flat 3D textures to represent the simulation grid [HBSL03] (see Figure 3.4a). Miyazaki et al. presented two methods for simulating cloud dynamics that were based on Stam's stable solver. First, they used so-called coupled map lattice (CML) which is an extension of cellular automaton. In CML, simulation space is divided into multiple lattices. Each lattice has several state variables which are, as opposed to regular cellular automaton, real numbers [MYDN01] (see Figure 3.4b). Later, Miyazaki et al. proposed a method that extended simulation of smoke by Fedkiw and Stam [FSJ01] by including phase transitions and adiabatic cooling [MDN02] (see Figure 3.4c).

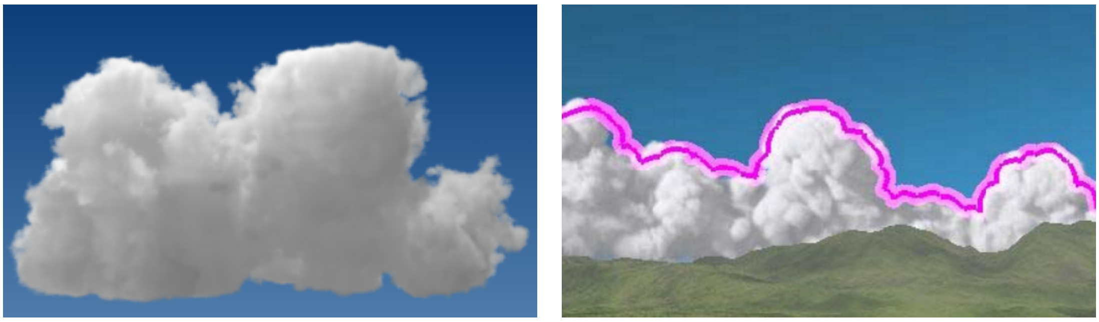
Geist et al. have proposed a novel technique for lighting participating media based on the Lattice Boltzmann method (LBM). The LBM is utilized as a grid-based photon transport model that uses Henyey-Greenstein phase function for anisotropic scattering of light [GRWS04] (see Figure 3.4d). They later expanded upon this approach by using two lattices, one is used for an improved photon mapping function while the second is used to



(a) [HBSL03] (b) [MYDN01]



(c) [MDN02] (d) [GRWS04]



(e) [GSW07] (f) [DYN06]

Figure 3.4: Physically-based methods.

generate spatial and temporal distribution of water density [GSW07] (see Figure 3.4e). These methods, at the time of their conception, provided sub-minute execution times on commodity hardware according to the authors.

One caveat of physically-based methods is the customizability of results by the user. Generally, the methods adhere to physical laws and do not give much freedom in customizing the behavior of fluid and therefore the resulting images. To overcome this problem, Dobashi et al. proposed a method that allows users to specify a desired silhouette of the final cloudscape [DYN06]. Their method uses the simulation proposed by Miyazaki et al. [MDN02] and renders the clouds using the method proposed by Dobashi et al. [DKY$^+$00] from 2000. The method has two key features that control the final shape: feedback control and geometric potential field. The feedback controller promotes growth of the cloud until top of the target shape is reached. This is done through controlling the latent heat and by supplying additional water vapor to the simulator. The geometric potential field generates external forces that prevent the clouds from growing outside the target shape [DYN06]. Results of this method with the pink target contour are shown in Figure 3.4f.

3.2 Procedural Methods

Procedural methods mainly focus on visually reproducing natural phenomena using functions, usually noise functions, or physical approximations. These methods are much less computationally expensive and can be utilized in real-time applications. Nowadays, procedural methods can be used to produce very realistic results at low costs. Their caveat is the need for trial and error customization of the simulation parameters to obtain desired results. In some ways, this can be considered as a strength when implemented correctly and used by artists who have understanding of the system. In these cases, the simulator can be a powerful tool for authoring cloudscapes according to artistic vision of their users. Duarte describes three main classes of procedural methods: cellular automata (CA) based methods, particle systems and volumetric techniques.

3.2.1 Cellular Automata

Nagel and Raschke [NR92] proposed a cellular automaton as a discrete 3D grid where each cell had three binary state variables: *hum*, *cld*, *act*. These denote humidity (water vapor), cloud, and state transition between water vapor and clouds. The simulation is then run by simple state transition rules. Main drawback of this method is that it does not take extinction into account, meaning that once a lattice node had state variable *cld* equal to 1, it would never revert back to 0. This was solved by Dobashi et al. [DNO98] by introducing an extinction state variable *ext* and new set of rules (see Figure 3.5a). Since this approach generated repeating cloud patterns, Dobashi et al. [DKY$^+$00] further improved the method by adding probability of extinction to generate more natural animations. This probability uses a simple coin flip approach where random number is generated when extinction of cloud should occur. When the random number is below the probability threshold, extinction occurs as in the previous method. In the same article, Dobashi et al. propose an efficient rendering technique of clouds and light shafts (see Figure 3.5b) which was later expanded by Harris in his thesis [Har03] by adding multiple forward scattering and anisotropic scattering (see Figure 3.5c).

3.2.2 Particle Systems

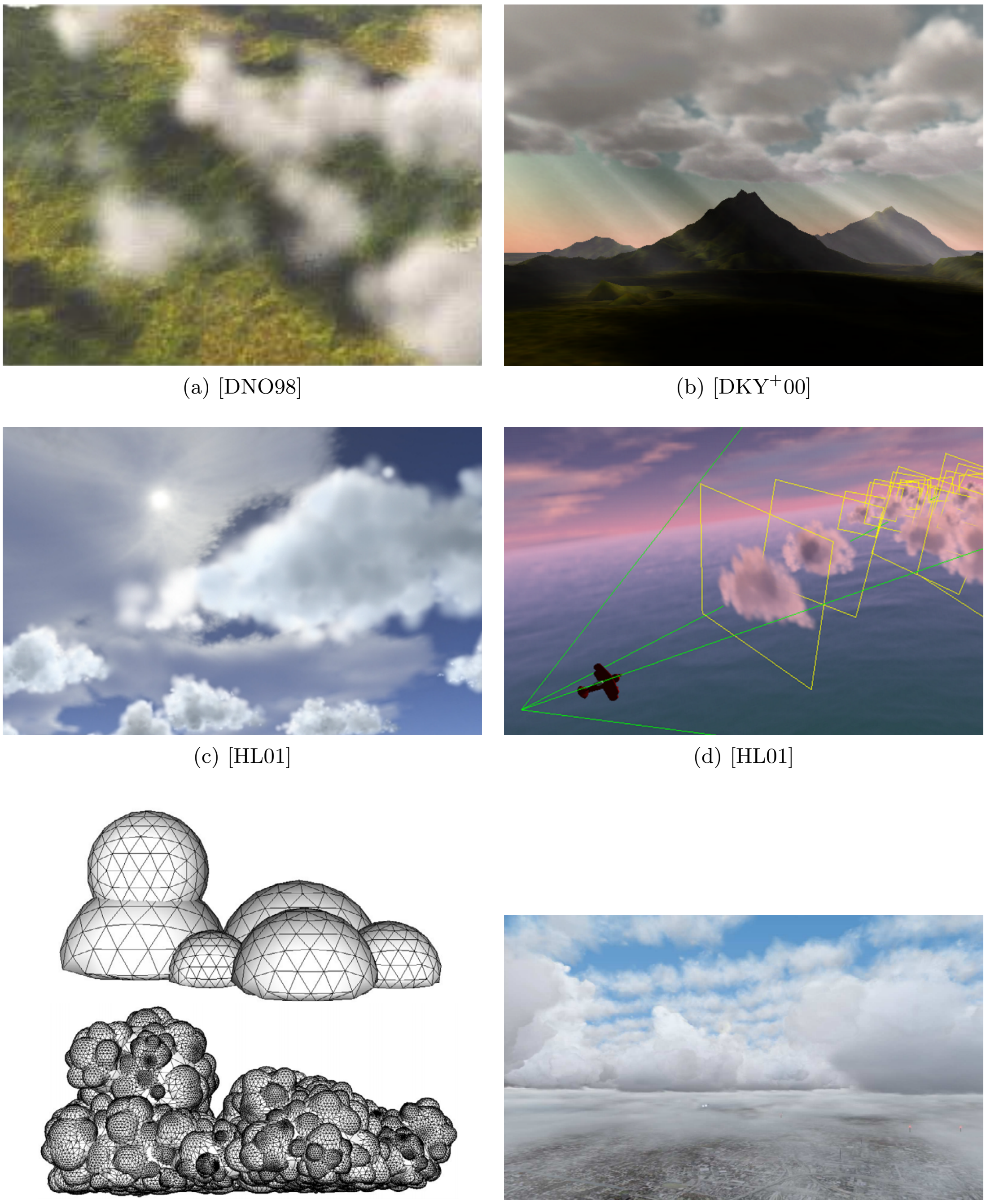
Particle systems are well-known and widely used due to their ability to simulate complex phenomena with boundaries that are hard to define. Particle systems consist of large numbers of particles that have multitude of properties such as position, lifetime, velocity, color, size and many more depending on their usage. They are widely used in the video game industry, where complex effects such as smoke, fire, sparks and many others are easily simulated with a predefined set of rules. Generally, particles are visualized as simple quads that may sometimes be oriented towards the camera, in which case they are called billboards. Naturally, particle systems are a great candidate for simulating and rendering clouds due to their ease of use and speed on modern hardware.

Harris used particle systems defined by users for cloud rendering using his multiple forward scattering method [HL01]. The method assumed static clouds and computed lighting using pixel read back in the preprocessing step. Harris also used impostors to draw the clouds in real-time. The method additionally took intersections of objects and impostors into account by splitting the intersected impostors into multiple layers (see Figure 3.5c and Figure 3.5d).

Wang [Wan03] further improved the method for Microsoft Flight Simulator: A Century of Flight by rendering a ring of 8 impostors into which numerous distant clouds are projected. The method achieved a 100x speedup over [HL01] and it was accompanied by a shading model that gave a high degree of artistic control to its users (see Figure 3.5f). A script for generating cloud textures from user defined boxes was proposed a year later by Wang [Wan04] to give artists an easy to control cloud authoring tool Figure 3.6a.

Bouthors and Neyret proposed a cumulus cloud shape generator based on particles and implicit surfaces [BN04]. The cloud model consists of levels that are iteratively generated. Each level contains a set of particles and the surfaces that are defined by them. During the iterative process, particles are generated in such a way that they repulse each other and populate all available free surface from the previous level. When a surface is overcrowded, particles are sometimes removed as well (see Figure 3.5e).

More recently, Yusov [Yus14] proposed a rendering algorithm for point sprites that uses precomputed light scattering for a set of possible light positions as shown in Figure 3.6b. Thanks to this, Yusov's method can handle dynamic lighting at real-time speeds. The precomputed data is stored in float look-up tables and is then utilized at runtime. For particle modeling/generation, Yusov uses a grid-based approach centered around camera with level of detail computed from its distance to a given particle, thanks to which large cloudy areas are handled with ease (for results see Figure 3.6c).



(a) [DNO98]

(b) [DKY$^+$00]

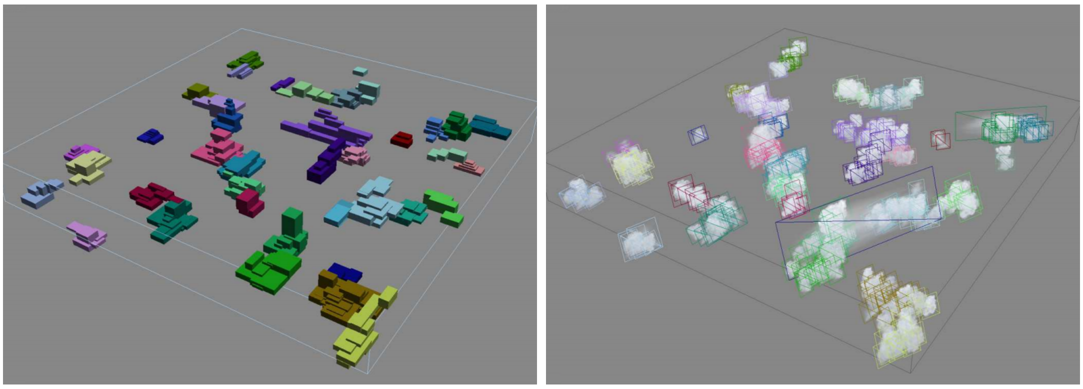
(c) [HL01]

(d) [HL01]

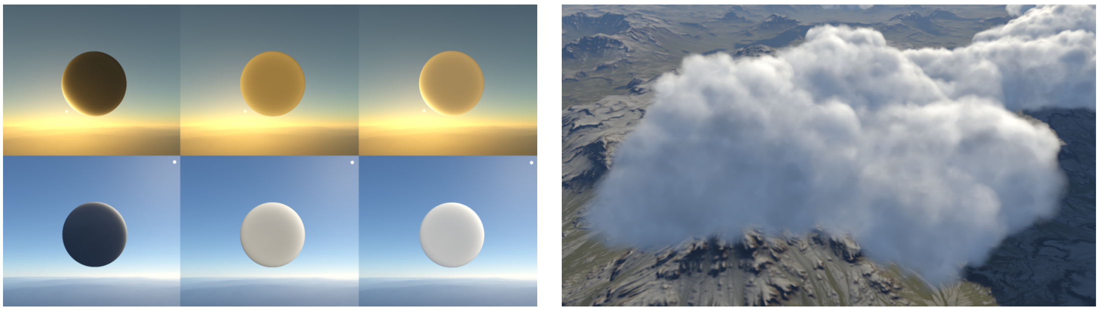
(e) [BN04]

(f) [Wan03]

Figure 3.5: Procedural methods that use particle systems.



(a) [Wan04]



(b) [Yus14]

(c) [Yus14]

Figure 3.6: Procedural methods that use particle systems (*continued*).

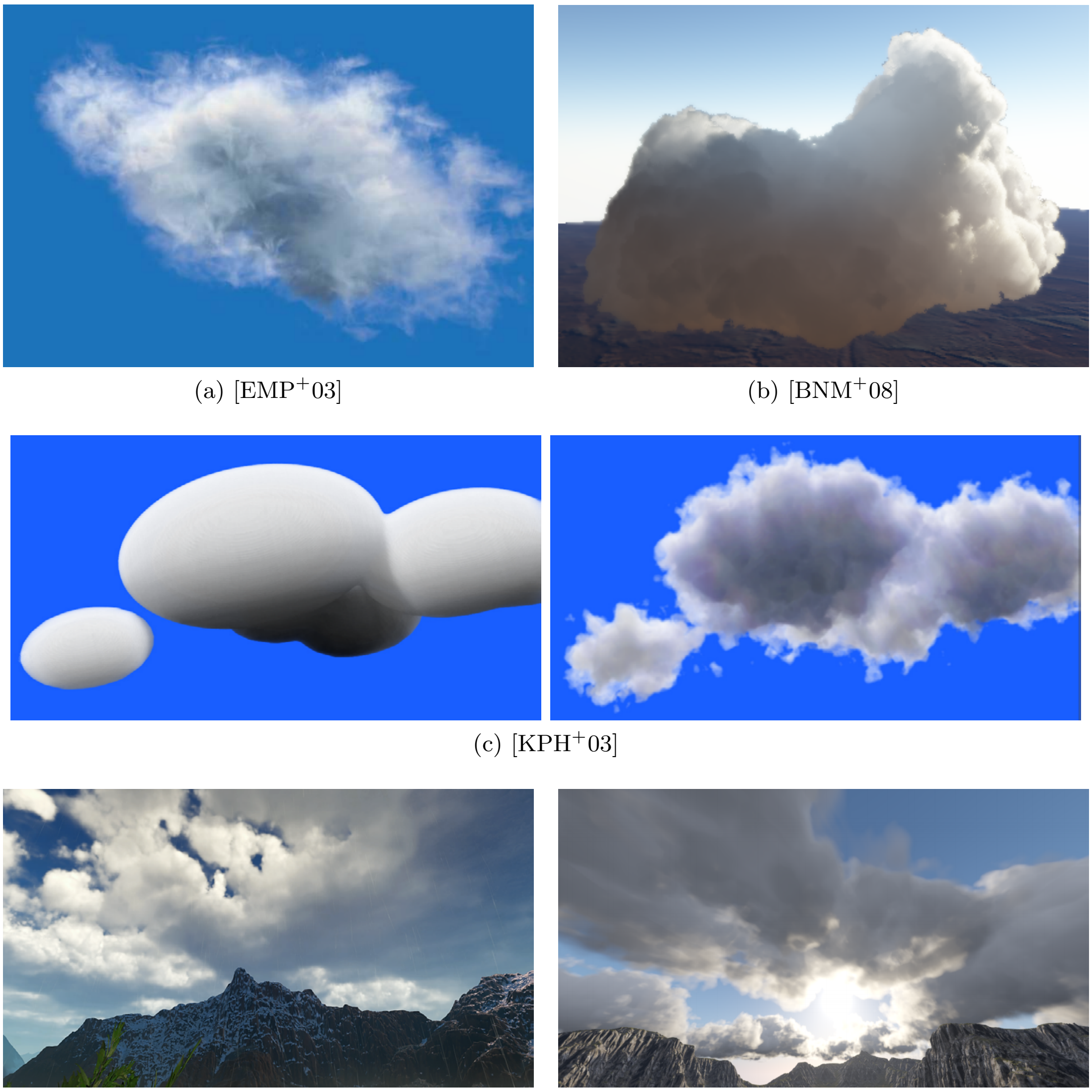
3.2.3 Volumetric Techniques

In the words of Ebert et al. [EMP$^+$03], "*Volumetric procedural techniques have all the advantages of procedural techniques and are a natural choice for cloud modeling because they are the most flexible, advanced modeling technique.*" The authors propose dividing the cloud model into two levels: its macrostructure and microstructure. Implicit functions and turbulent volume densities model these, respectively [EMP$^+$03]. Procedural noise and turbulence functions create the cloud's microstructure while implicit functions such as spherical or elliptical implicit primitives are used to create the overall shape (see Figure 3.7a). This method inspired multiple researchers. It provided a basis for the method of Kniss et al. [KPH$^+$03] where frequency noise function was used to generate iridescence effects when rendering clouds as shown in Figure 3.7c.

Another notable method was proposed by Bouthors et al. that renders cumulus clouds using a complex light transport analysis inside the cloud volume. Their method takes advantage of both volumes and surfaces by representing the cloud boundary with a regular triangular mesh while the high-frequency density variations are sampled from a Hypertexture [PH89]. Hypertexture is used to generate complex 3D textures such as fur or erosion by decomposing them into three regions: *hard region* where object is solid, *soft region* where density of the object varies, and *outside region* where the object does not exist. Bouthors et al. observe and compute light scattering for multiple orders starting with order 1 (single scattering) and continuing with eight sets of orders in range [2, inf). For single scattering, the method analytically integrates the Mie single scattering along eye direction while for multiple scattering a complex collection method is used. Their approach provided interactive frame rates in some cases depending on the mesh and whether the Hypertexture is evaluated on the fly. Result of the method is shown in Figure 3.7b.

Recently, a volumetric approach to modeling and rendering procedural clouds has been heavily used in the game industry. This novel approach was first used in Guerilla Games' Horizon Zero Dawn and it was first presented by Schneider at SIGGRAPH [SV15]. The method uses adaptive ray marching that samples two 3D and one 2D noise textures. More specifically, a combination of Worley's and Perlin's noise is used when generating the 3D textures. For rendering of the clouds, anisotropic scattering is simulated by using a proposed composition of Beer's law and a powder effect, coined as Beer's-powder approximation method. Furthermore, the Henyey-Greenstein phase function is used to approximate anisotropic scattering. With optimizations that reproject last frame and use low resolution buffers on edges of screen, the method renders in 2ms on the Playstation 4 console according to the authors. The system is constantly being updated and now has an official name - Nubis [SV17]. We present its examples in Figure 3.2a and Figure 3.7d.

This method has gained a lot of traction and is generally used in other studios. As an example, Hillaire [Hil16] shows that Frostbite engine, which powers a wide range of Electronic Arts titles, uses the same approach with some minor modifications that usually affect cloud authoring and weather controls (see Figure 3.7e). For more in-depth analysis of this method, we would like to refer the reader to Häggström's thesis [Häg18].



(a) [EMP$^+$03]

(b) [BNM$^+$08]

(c) [KPH$^+$03]

(d) [SV15]

(e) [Hil16]

Figure 3.7: Volumetric procedural methods.

3.2.4 Image-Based Methods

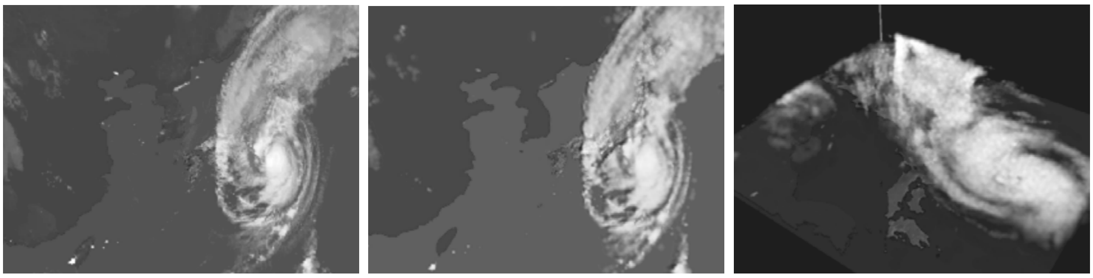
A special subcategory of procedural methods are so-called image-based methods. These are sometimes listed separately as is the case in Dobashi's survey [DIYN17]. Image-based methods generate clouds by trying to replicate visual properties or shapes from given input images. They can be divided into two groups: satellite image-based and regular photograph-based methods.

Satellite image-based methods generate large areas of clouds as seen from outer space. Dobashi et al. [DNYO98] generate clouds represented by metaballs by simply determining whether a given image pixel is or is not a cloud pixel. After that, each metaball is given properties so that the final image resembles the input image as closely as possible (see Figure 3.8a).

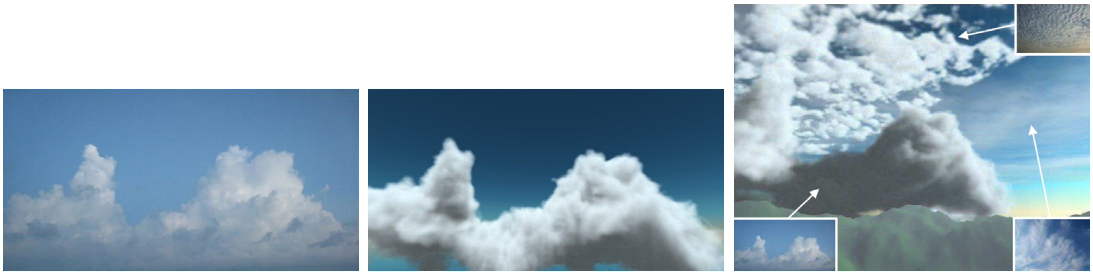
From photograph-based methods, let us present two different approaches by Dobashi et al. that focus on cloud modeling and rendering, respectively. The first method [DSY10] uses input photographs to generate three types of clouds: cirrus, altocumulus and cumulus; where each has different representation. Two-dimensional textures are used for cirrus clouds, metaballs for altocumulus clouds, and volumetric approach is used for cumulus clouds. The texture and density distributions are then generated using the input images as shown in Figure 3.8b. The second method [DIO^{+}12] solves an inverse rendering problem where, given an input photograph and an existing cloud density distribution, the parameters needed for rendering clouds are estimated. Genetic algorithms and histograms are utilized for searching physically correct parameters such that the appearance of the rendered clouds is visually similar to the clouds in the input photograph. Results of this method are presented in Figure 3.8c.

3.3 Closing Remarks

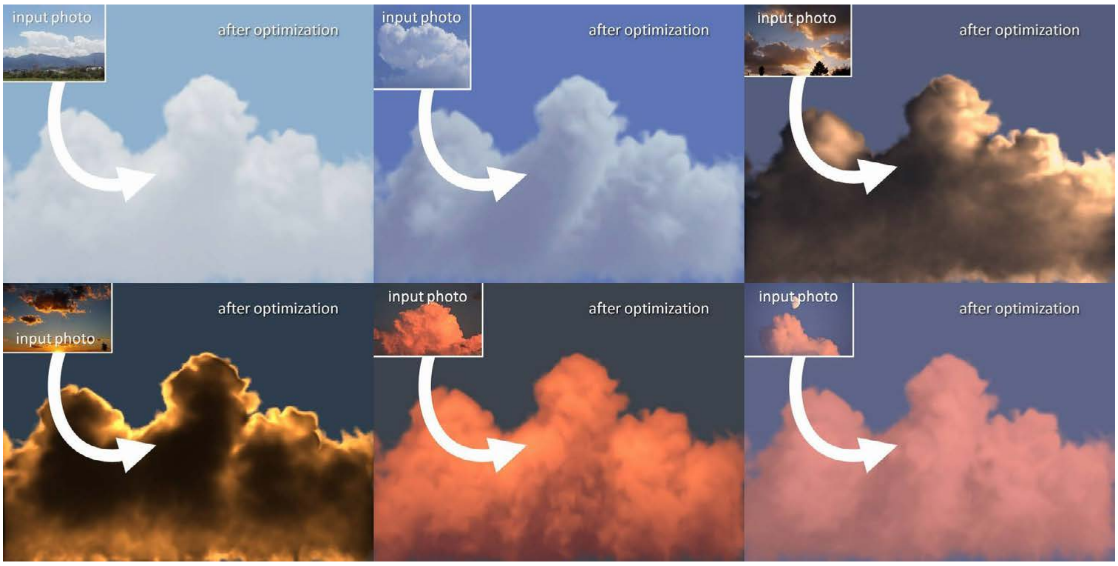
Cloud simulation is an important topic of computer graphics with a vast array of proposed methods. Since we have described only a select few, we would like to refer the reader to Duarte's survey for more in-depth examination of the field [Dua16]. Nice overview of multiple methods, especially image-based methods, can also be seen in Dobashi's survey [DIYN17].



(a) [DNYO98]



(b) [DSY10]


input photo
after optimization
input photo
after optimization
input photo
after optimization
input photo
after optimization
input photo
after optimization
input photo
after optimization

(c) [DIO$^+$12]

Figure 3.8: Image-based methods.

4 Cloud Simulation Using SkewT/LogP Diagrams

In his PhD thesis, Duarte proposes a novel approach to simulating clouds, more specifically, cumulus clouds in Chapter 3 and orographic clouds in Chapter 4 [Dua16]. This novel approach falls into physically-based methods with one exception. It solves Navier-Stokes equations explicitly using sounding data that are obtained by meteorological stations daily. This data is freely available online, especially in the case of the United States of America with websites such as `www.twisterdata.com` and many others. These soundings contain enough information for us to solve equations of cloud motion using well-known line-to-line intersection algorithms when plotted in so-called SkewT/LogP diagrams.

4.1 SkewT/LogP Diagram

SkewT/LogP diagram is a commonly used meteorological chart on which temperature, pressure, density, mixing ratio, wind speed, and many other properties of the atmosphere are plotted for a single point of Earth's surface. SkewT/LogP diagram is the core of Duarte's novel approach since it enables us to solve Equation 2.9 by explicitly solving Equation 2.8. This is done through determining three important parameters that are featured in the simulation, namely: convective temperature T_c, convective condensation level CCL, and equilibrium level EL.

The convective temperature T_c is a temperature at which particles of an air parcel start to rise from the Earth's surface. CCL is a level at which water vapor starts to condense out of dry air, i.e. first point at which a cloud starts to form, whereas EL is a level at which particles of a forming cloud stop rising as shown in Figure 4.1.

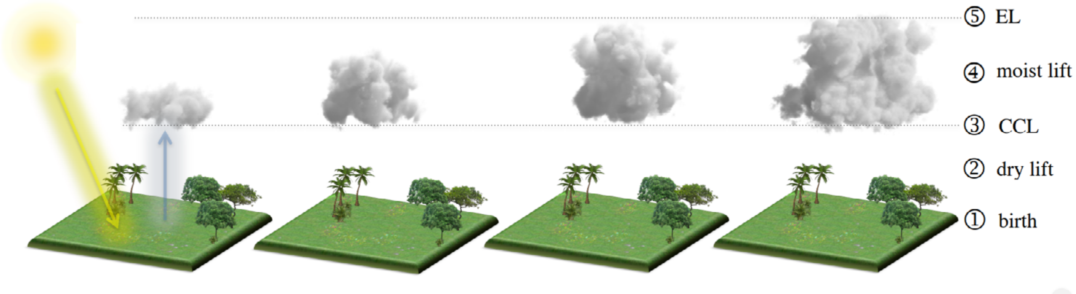


Figure 4.1: Individual simulation stages of [Dua16].

All these parameters can be obtained by intersecting individual curves of the diagram. These curves can be categorized into two groups: main curves and sounding curves. Main curves are not dependent on the input sounding data as opposed to the sounding curves which differ from sounding to sounding.

4.1.1 Main Curves

Isobars

Isobars are drawn as horizontal lines that depict a level of constant pressure. In the current implementation, they are traced for every sounding data row which is 25mb = 25hPa in a vertical logarithmic scale. Please note that we use normalized coordinates for rendering the diagram, meaning that values of x and y are in range $[0, 1]$. Isobar for a specific pressure point P can be computed using

$$y = \frac{\lg(P) - \lg(P_{min})}{\lg(P_{max}) - \lg(P_{min})} \tag{4.1}$$

This results in vertically mirrored coordinates in OpenGL. This problem can be alleviated by using orthographic projection with reversed y axis. Note that $\lg(P) = \log_{10}(P)$.

Isotherms

Isotherms are skewed straight lines with a 45 degree slope. The mapping of points in our current system is a little different than the one proposed in Duarte's thesis. Isotherm coordinates for given temperature T are simply computed by using

$$T_{normalized} = \frac{T - T_{min}}{T_{max} - T_{min}} \tag{4.2}$$

$$x = T_{normalized} + 1 - y \tag{4.3}$$

Individual mapping functions are shown in Listing 8.1.

Isohumes (Mixing Ratio Lines)

Isohumes are lines of equal mixing ratio that describe the saturation mixing ratio of air. Saturation mixing ratio $w(T, P)$ is the maximum amount of water vapor that air can hold for given pressure and temperature. Isohumes therefore relate the mass of water vapor in a parcel (in g) to the mass of dry air (in kg) as follows:

$$w(T, P) = \frac{\varepsilon \cdot e(T)}{P - e(T)} \tag{4.4}$$

where $\varepsilon = R_d/R_m \approx 0.622$ ($R_d = 287.05307$, $R_m = 461.5$) and $e(T)$ stands for the saturation vapor pressure that can be approximated by the formula

$$e(T) \approx C \exp\left(\frac{A \cdot T}{T + B}\right) \tag{4.5}$$

where $-30°\text{C} \leq T \leq 35°\text{C}$, $A = 17.67$, $B = 243.04$ and $C = 611.2$. Note that the formula expects temperature in °C and returns pressure in pascals [Bol80].

Given a constant value for the mixing ratio $w(T, P) = W$, we can express T in terms of P as follows:

$$T(P) = \frac{B \ln\left(\frac{W \cdot P}{C(W+\varepsilon)}\right)}{A - \ln\left(\frac{W \cdot P}{C(W+\varepsilon)}\right)} \tag{4.6}$$

This equation is then used to determine an isohume that passes through point (T, P) by calculating the temperature $T(P + \delta)$, with δ being a small integer value. Therefore, pair of points (T, P) and $(T(P + \delta), P + \delta)$ define the final curve.

Dry Adiabats

Dry adiabats are the $1/\log(P)$ curves (e.g. highlighted gray curve in Figure 4.3). These represent the movement of particles between ground and CCL, i.e. movement before the particles become saturated. This curve can be obtained from

$$T(P) = \theta \Bigg/ \left(\frac{P_0}{P}\right)^{R_d/c_p} \tag{4.7}$$

where P_0 is the ground pressure, P is a pressure for which we want to obtain the absolute temperature, and θ is the potential temperature. It is important to note that all these computations need to be done with SI units, therefore θ needs to be first converted to K and then the final temperature $T(P)$ is converted back to °C for use in the diagram. For the actual implementation of the dry adiabat creation, we would like to refer the reader to Listing 8.2.

Moist Adiabats

Moist adiabats, also known as pseudoadiabats, give us information about the upwards motion of saturated air, i.e. air that has reached its convective level. As opposed to dry adiabats, moist adiabats cannot be created by non-iterative approach. This means that the absolute temperature T_f at any final pressure P_f cannot be solved directly knowing an initial pressure and absolute temperature P_s and T_s, respectively [BS13]. The procedure to create a pseudoadiabat therefore requires iterative approach with small step of ΔP from an initial given point to find the temperature T_f at final pressure P_f. The saturated adiabatic lapse rate (SALR) $\Gamma_s = -dT/dz$ is defined as

$$\frac{\Gamma_s}{\Gamma_d} = \frac{1 + \frac{L_v(T)w(T,P)}{R_d T}}{1 + \frac{L_v^2(T)w(T,P)\varepsilon}{R_d c_{pd} T^2}} \tag{4.8}$$

where $\Gamma_d \approx g/c_{pd} \approx 9.76\,[°\text{C}\,\text{km}^{-1}]$ is the dry adiabatic lapse rate (DALR), $c_{pd} = 1005.7$ is the specific heat at constant pressure for dry air, $w(T, P)$ is the saturation mixing ratio and $L_v(T)$ stands for the latent heat of vaporisation/condensation and is given by:

$$L_v(T) = (aT^3 + bT^2 + cT + d) \cdot 1000 \tag{4.9}$$

where $a = -6.14342 \cdot 10^{-5}$, $b = 1.58927 \cdot 10^{-3}$, $c = -2.36418$ and $d = 2500.79$ [Dua16]. It is usually approximated by its value for 0°C where $L_v \approx 2.501 \cdot 10^6\,\mathrm{J\,kg^{-1}}$. We can rewrite Equation 4.8 in the terms of pressure as follows [BS13]:

$$\frac{dT}{dP} = \frac{1}{P}\frac{R_d T + L_v(T)w(T,P)}{c_{pd} + \frac{L_v^2(T)w(T,P)\varepsilon}{R_d T^2}} \tag{4.10}$$

This equation is then utilized when creating the moist adiabat iteratively as shown in Listing 8.3. Similarly, Duarte describes moist adiabats using the pseudoadiabatic lapse rate based on the AMS Glossary of Meteorology [ams12] as

$$\Gamma_s(T,P) = g\frac{(1+w(T,P))\left(1+\frac{L_v(T)\cdot w(T,P)}{R_d\cdot T}\right)}{c_{pd} + w(T,P)\cdot c_{pv} + \frac{L_v(T)^2\cdot w(T,P)\cdot(\varepsilon + w(T,P))}{R_d\cdot T^2}} \tag{4.11}$$

where $w(T,P)$ is the mixing ratio of water vapor, $c_{pd} = 1005.7$ and $c_{pv} = 1875$ are the specific heat of dry air and the specific heat of water vapor at constant pressure, respectively. The pressure lapse rate is given by

$$\frac{dT}{dP} = \frac{\Gamma_s(T,P)}{\rho\cdot g} \tag{4.12}$$

Therefore, to obtain next temperature $T(P_1)$ and pressure P_1 from initial temperature $T(P_0)$ and pressure P_0, we need to integrate the pressure lapse rate as follows:

$$T(P_1) = T(P_0) + \int_{P_0}^{P_1}\frac{\Gamma_s(T,p)}{\rho\cdot g}dp \tag{4.13}$$

where density ρ is given from the ideal gas law:

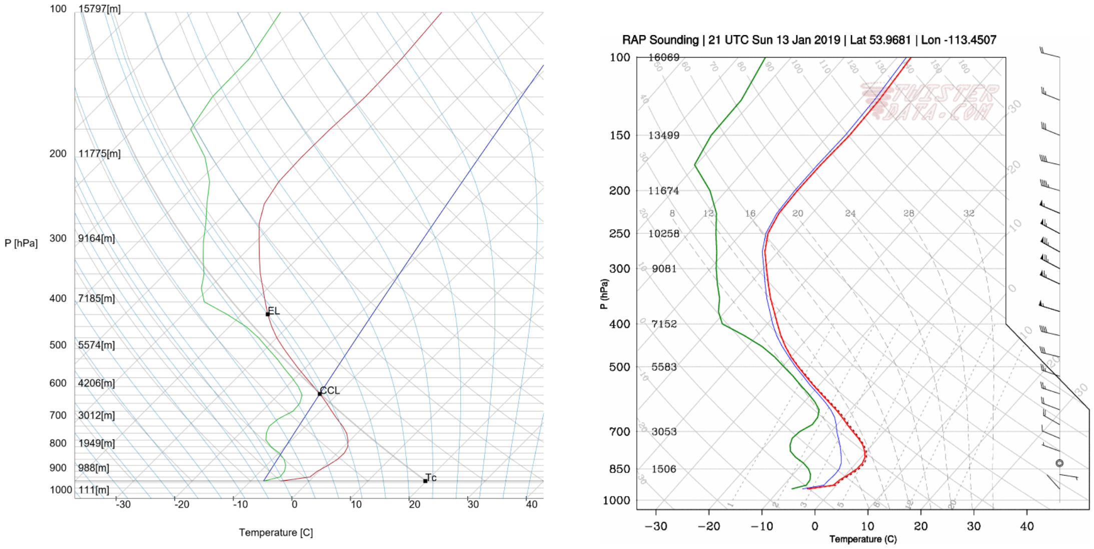
$$\rho = \frac{P - e(T)}{R_d\cdot T} + \frac{e(T)}{R_m\cdot T} \tag{4.14}$$

4.1.2 Sounding Curves

Part of the SkewT/LogP diagrams are, of course, the sounding curves obtained by radiosondes. These are represented as piece-wise linear curves in the diagram. We are mainly interested in three parameters when it comes to plotting data: ambient temperature (TEMP, in °C), dew point temperature (DWPT, in °C) and the mixing ratio (MIXR, in g/kg). Another important parameters are mainly the wind direction (DRCT, in degree) and the wind speed (SKNT, in knots) which are useful when simulating wind. Example of sounding data is presented in Table 4.1 and its corresponding SkewT/LogP diagrams (ours and reference) are shown in Figure 4.2.

PRES [hPa]	HGHT [m]	TEMP [°C]	DWPT [°C]	RELH [%]	MIXR [g/kg]	DRCT [deg]	SKNT [knot]	TWTB [°C]	TVRT [°C]	THTA [K]	THTE [K]	THTV [K]
943.0	673	-4.9	-7.8	80	2.3	318	1	-5.8	-4.5	272.8	279.2	273.2
925.0	829	-0.6	-5.7	68	2.7	97	5	-2.4	-0.2	278.6	286.4	279.1
900.0	1048	-0.3	-5.9	66	2.7	102	5	-2.3	0.1	281.1	289.1	281.6
875.0	1274	0.5	-6.7	58	2.6	99	5	-2.1	0.9	284.3	292.0	284.7
850.0	1506	1.1	-8.0	50	2.5	102	2	-2.2	1.5	287.3	294.7	287.7
825.0	1746	1.2	-9.9	43	2.2	299	0	-2.8	1.6	289.9	296.5	290.2
800.0	1994	0.7	-12.4	36	1.9	293	4	-3.8	1.0	291.8	297.6	292.1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
100.0	16069	-57.4	-84.9	1	0.0	284	22	-58.3	-57.4	416.3	416.3	416.3

Table 4.1: Example of sounding data obtained from `www.twisterdata.com` that corresponds with the SkewT/LogP diagrams shown in Figure 4.2.



(a) Our current diagram visualization.

(b) Diagram from `www.twisterdata.com`.

Figure 4.2: Comparison of implemented SkewT/LogP diagram with original source.

4.2 Simulation Steps

The simulation can be decomposed into three main steps: birth, dry lift and moist lift. The important idea is that we are able to solve the presented equations of cloud motion by using line-to-line intersections of the sounding curves with main curves of the STLP diagram.

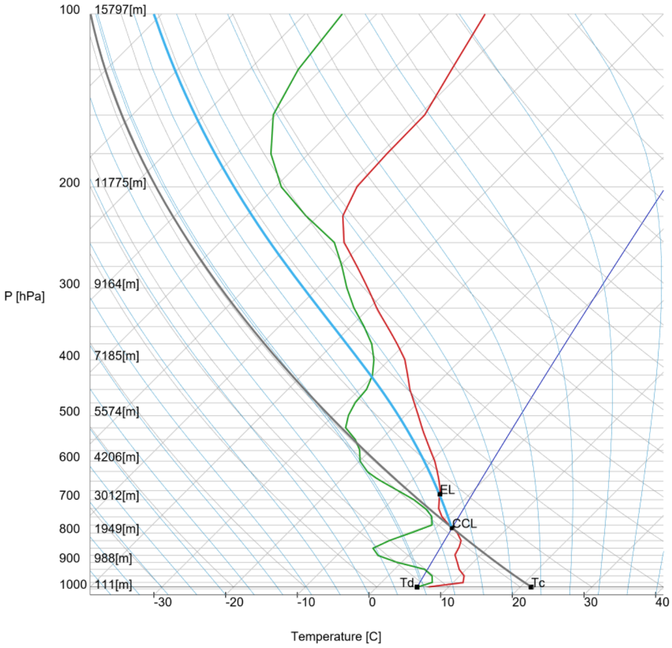


Figure 4.3: SkewT/LogP diagram with highlighted curves.

4.2.1 Birth

In Duarte's thesis, the process of creating particles is described as gestation (or generation) followed by their parturition (birth). In layman's terms, this means that we first generate particles on the ground (randomly, in an area) and then wait for them to reach the convective temperature T_c by slowly heating them up. When the temperature of particles reaches T_c, they begin to rise above the ground in the form of dry air. Thus, the main variable we need to compute to simulate the generation and parturition of particles is the convective temperature T_c. The computation consists of the five following steps:

1. Get the dew point (T_d, P_0) denoted as T_d in Figure 4.3. We know this from the sounding data column DWPT (dew point temperature). You may also notice that it is an initial point of the piece-wise linear curve representing the dew point temperature which is higlighted with a green color in the diagram.

2. Create the mixing ratio line that passes through the dew point (T_d, P_0). Shown as a dark blue line in Figure 4.3.

3. Find the CCL point denoted as $(T_{\mathrm{CCL}}, P_{\mathrm{CCL}})$. It is the intersection of the mixing ratio line from the previous step and one of the line segments of the piece-wise linear curve C_a relative to ambient temperature (red in the diagram). CCL can be therefore found using a line-to-line intersection algorithm.

4. Compute potential temperature θ at the CCL which is given by

$$\theta_{\mathrm{CCL}} = T_{\mathrm{CCL}} \left(\frac{P_0}{P_{\mathrm{CCL}}} \right)^{R_d/c_{pd}} \tag{4.15}$$

5. We can use the potential temperature θ_{CCL} to obtain the convective temperature T_c. This can be easily done by plotting a dry adiabat with θ_{CCL} from the previous step and finding its absolute temperature T_0 for ground pressure P_0 where $T_0 = T_c$. This dry adiabat is shown as a wide dark grey curve in Figure 4.3.

Duarte shows that using a limited number of prepared profiles for particles in an air parcel should be utilized to obtain believable results. This is done by precomputing all important curves and values (dry and moist adiabats, CCL, EL) for each particle profile which will be then used in the simulation. Particles are to be generated randomly in predefined areas on the ground. The profiles are determined by T_c from the sounding data in such a way, that they have a convective temperature in a predefined range $[T_c, T_c + \Delta T_c]$. Since computing a custom set of curves for each particle π_i is unfeasible, particles are grouped into these so-called profiles. Each profile Π_k uses its own set of curves that are determined by its convective temperature T_{c_k}. This means that profile Π_k has a convective temperature $T_{c_k} = T_c + k \cdot \frac{\Delta T_c}{N}$ where N denotes the number of profiles we want to use.

4.2.2 Dry Lift and Moist Lift

The second and third steps of the simulation are based on solving the equation

$$\frac{dv}{dt} = g \frac{\theta_p(z) - \theta_a(z)}{\theta_a(z)} \tag{4.16}$$

from which the rising force $\vec{F}$ as well as velocity $\vec{v}$ for a particle are computed. We present both these steps at once since they differ in a small detail only. During the dry lift, particles advect from the ground to CCL, then, during the moist lift, they advect from CCL to EL. The trajectories for both stages are shown in Figure 4.4.

To solve the Equation 4.16 for particle π_i at any point (T, P) that is located at altitude z_i, we need to determine the potential temperature of the particle θ_i and the ambient potential temperature of its surrounding environment θ_a. The value of θ_i is given by an intersection of the isobar particle π_i lies on with either its dry or moist adiabat for dry lift and moist lift, respectively. Similarly, the ambient potential temperature of the environment θ_a is simply given as the intersection of the same isobar and the ambient temperature sounding curve.

When these intersections are obtained, we can compute Equation 4.16 to get acceleration of the particle π_i. With acceleration, we can easily compute the vertical velocity v_i and the vertical displacement Δz using basic kinematic equations.

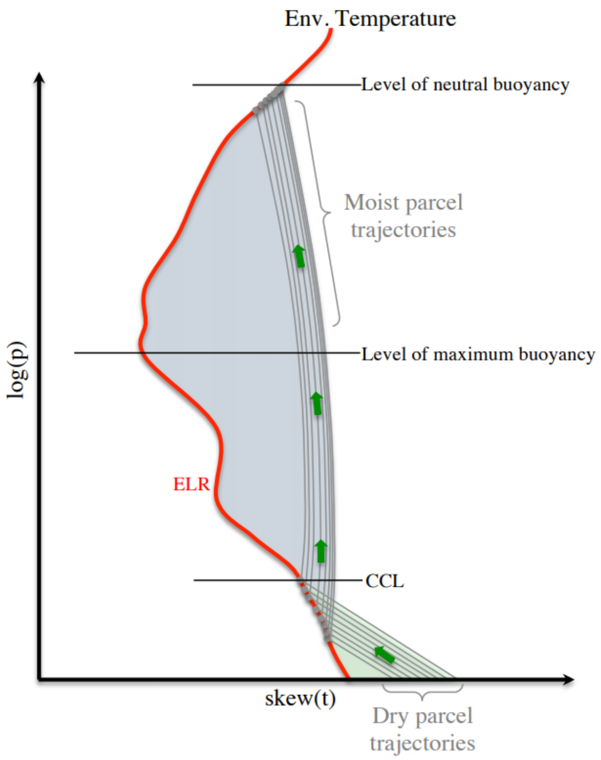

Env. Temperature
Level of neutral buoyancy
Moist parcel trajectories
Level of maximum buoyancy
log(p)
ELR
CCL
skew(t)
Dry parcel trajectories

Figure 4.4: Trajectories of particles for dry and moist lift. Note the usage of multiple profile curves for different convective temperatures T_c [Dua16].

Pressure Derivation

To use altitude and pressure interchangeably, a direct derivation of pressure P in hectopascals [hPa] based on altitude z in meters [m] is used. It is defined as [qui04]:

$$P = \left(\frac{44331.514 - z}{11880.516}\right)^{1/0.1902632} \tag{4.17}$$

From this, we can also determine inverse relation as:

$$z = 44331.5 - 4946.62 \cdot (P \cdot 100)^{0.190263} \tag{4.18}$$

This is useful since we do not want to find the correct value from the sounding data which would require a binary search and an interpolation at each conversion.

Wind

Duarte's method uses the sounding data to simulate wind as well. There are two main properties of wind that can be read from the sounding data table: the wind direction $\alpha \in [0, 360]$ (DRCT, in degrees, clockwise from true north) and wind speed r (SKNT, in knots). From this we can obtain particle displacement on x and y axes (in their respective altitude as defined in the sounding data table) as

$$\Delta x = r' \cos(\alpha) \tag{4.19}$$

$$\Delta y = r' \cos(\alpha) \tag{4.20}$$

where r' is the speed converted to meters per second, $r' \approx 0.514444 \cdot r$ [m/s]. In Algorithm 1 we present a pseudocode for the whole process of the described lift motion.

Algorithm 1: Dry and Moist Lift Motion

```
Data: $\Pi$                                                  ▷ set of all particles
Data: $C_a$                                                  ▷ ambient temperature curve
Result: Lift Motion
foreach $\pi_i \in \Pi$ do
    $P_i \leftarrow$ pressure at altitude $z_i$ of particle $\pi_i$      ▷ Equation 4.17
    $l \leftarrow$ isobar line at pressure $P_i$
    if $P_i > P_{\text{CCL}}$ then                           ▷ particle $\pi_i$ is below CCL
        $C_i \leftarrow$ dry adiabat for $\pi_i$
    else                                                     ▷ particle $\pi_i$ is above CCL
        $C_i \leftarrow$ moist adiabat for $\pi_i$
    $(T_A, P_i) \leftarrow (l \cap C_a)$
    $(T_B, P_i) \leftarrow (l \cap C_i)$
    $\theta_a \leftarrow$ potential temperature at $(T_A, P_i)$          ▷ Equation 2.8
    $\theta_i \leftarrow$ potential temperature at $(T_B, P_i)$          ▷ Equation 2.8
    $a \leftarrow 9.81 \cdot (\theta_i - \theta_a)/\theta_a$             ▷ Equation 4.16
    $v_i \leftarrow v_i + a \cdot t$
    $\Delta z \leftarrow v_i \cdot t + 1/2 \cdot a \cdot t^2$
    $z_i \leftarrow z_i + \Delta z$
    $x_i \leftarrow x_i + \Delta x$                                      ▷ Equation 4.20
    $y_i \leftarrow y_i + \Delta y$                                      ▷ Equation 4.20
    $P_i \leftarrow$ pressure at new altitude $z_i$                      ▷ Equation 4.18
```

4.3 Orographic Clouds

In the previous section, the main idea behind Duarte's cloud simulator was established, that is, the usage of sounding data and SkewT/LogP diagrams for simulating cumulus clouds. Now let us look at his proposition of using the same principles for simulating orographic clouds.

The only difference lies in the set of parameters that are needed for the simulation. Terrain as an obstacle has to be considered as well. In the previous method, we needed T_c, CCL and EL for the individual simulation steps. Here, we require the lifting condensation level (LCL), the level of free convection (LFC) and, once again, the equilibrium level (EL) as shown in Figure 4.5. Beware, that EL is now computed differently and it is not the same point as in the previous approach. Let us now look more closely at the process of convection for orographic clouds.

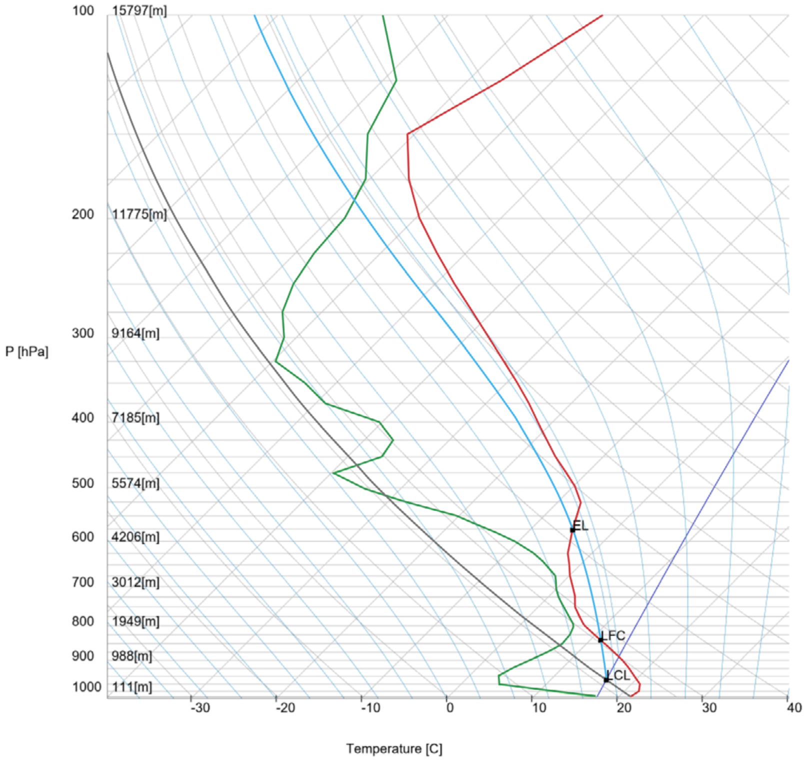


Figure 4.5: Diagram showing the orographic parameters used in Duarte's method.

Generally, a parcel of dry air is forced to move from low elevation to high elevation by wind and an obstacle (mountain). This can be shortly described as a forced convection because the air parcel by itself would not otherwise ascend in any manner. As it ascends, it cools dry-adiabatically while saturation is not reached. The point where saturation is reached (and therefore condensation starts) is called the lifting condensation level (LCL). After passing the LCL, the air has negative buoyancy and is forced to ascend moist-adiabatically due to orographic lift. When reaching the LFC, the air attains positive buoyancy and starts lifting moist-adiabatically on its own. The air parcel stops its ascension when reaching the EL similarly to the previous method. In short, the algorithm is the same bar the usage of different parameters and the forced ascension due to orographic lift.

The LCL is the intersection of the mixing ratio line which originates in the dew point T_d and the dry adiabat that passes through the ambient temperature sounding curve on the ground as described in Algorithm 2.

Algorithm 2: Computation of LCL point $(T_{\mathrm{LCL}}, P_{\mathrm{LCL}})$ for a particle π_i

```
$(T_d, P_0) \leftarrow$ dewpoint on the ground
$l \leftarrow$ mixing ratio line through $(T_d, P_0)$
$(T_i, P_0) \leftarrow$ ambient temperature of the particle $\pi_i$ on the ground
$C_d \leftarrow$ dry adiabat through $(T_i, P_0)$
$(T_{\mathrm{LCL}}, P_{\mathrm{LCL}}) \leftarrow l \cap C_d$
```

By plotting a moist adiabat through the LCL, we can find both the LFC and the EL. The LFC is the first intersection of said moist adiabat with the ambient temperature curve. The EL is simply the second intersection of these two curves. This is illustrated step by step in Algorithm 3.

Algorithm 3: Computation of the LCF and EL points for a particle π_i

```
$(T_{\mathrm{LCL}}, P_{\mathrm{LCL}}) \leftarrow$ LCL point of particle $\pi_i$
$C_m \leftarrow$ moist adiabat through $(T_{\mathrm{LCL}}, P_{\mathrm{LCL}})$
$(T_{\mathrm{LFC}}, P_{\mathrm{LFC}}) \leftarrow$ first intersection of $C_m$ and $C_a$
$(T_{\mathrm{EL}}, P_{\mathrm{EL}}) \leftarrow$ second intersection of $C_m$ and $C_a$
```

4.3.1 Terrain Influence

Besides the parameter points, only addition to the simulation is terrain influence. Duarte uses triangular meshes to describe the terrain. For determining how the particle moves around it, the algorithm checks whether any faces of the mesh are intersected by the velocity vector of the particle. If more than one face is intersected, the algorithm considers the visible face from particle's position only. The velocity vector $\vec{v}$ is then projected onto the face which is denoted by $\vec{v}_{proj}$ as shown in Figure 4.6.

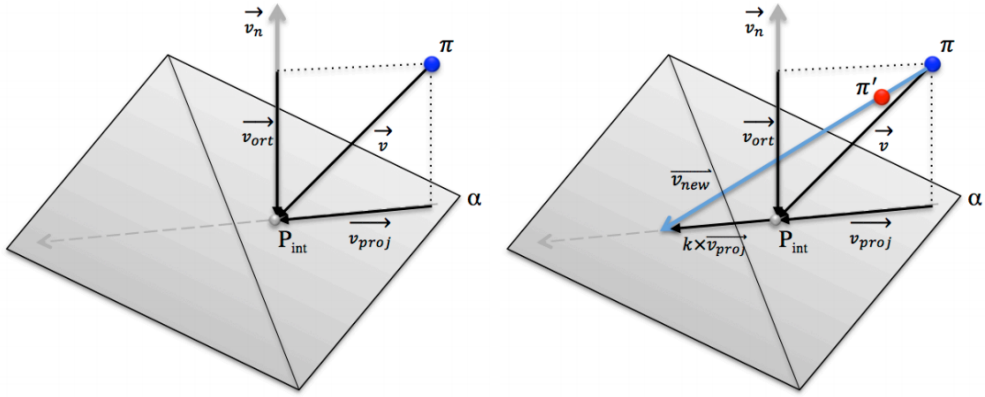


Figure 4.6: Obtaining $\vec{v}_{new}$ from the intersection point P_{int} and projected velocity $\vec{v}_{proj}$.

Vector $\vec{v}_{proj}$ represents the velocity parallel to the mountain face and therefore the influence of the terrain. Two additional factors are considered: the terrain roughness $R \in [0, 1]$ and the distance d of the particle to the terrain. Using these properties, an attenuation factor is defined as

$$k = \frac{(1 - R)}{d^2} \tag{4.21}$$

for each face of the mountain. Using the attenuation factor, the altered velocity vector $\vec{v}_{new}$ is then defined as

$$\vec{v}_{new} = k \cdot \vec{v} \tag{4.22}$$

The attenuation factor R determines terrain roughness. Terrain is without obstacles for small $R \approx 0$, while for large $R \approx 1$, the terrain is considered to be covered with obstacles such as trees.

5 Lattice Boltzmann Method

Second method that is used by the proposed simulator is the real-time Lattice Boltzmann method. Lattice Boltzmann method (LBM) is a fluid simulation method that falls under the set of computational fluid dynamics (CFD) methods. It solves the discrete Boltzmann equation as opposed to Navier-Stokes equations. The method was chosen for simulating wind as it is one of the areas in which Duarte's approach uses a naive solution. Since both the 2D and 3D versions of the LBM were implemented during the creation of this thesis, both will be described. Note that only the 3D version is available in the final framework while the 2D version is provided as a standalone project.

The main idea of LBM is that fluids can be perceived as a large number of very small particles interacting with each other, exchanging energy, colliding. In other words, fluids can be perceived as a group of their molecules on a microscopic level. LBM simplifies this model by representing these molecular particles in an equidistant grid of nodes (sometimes named cells or sites) called the lattice. Each node contains a distribution function $f_i(\vec{x}, t)$ that describes the probability that a particle in node $\vec{x}$ will move in the direction $\vec{e_i}$ or not move at all at time step t as shown in Figure 5.1.

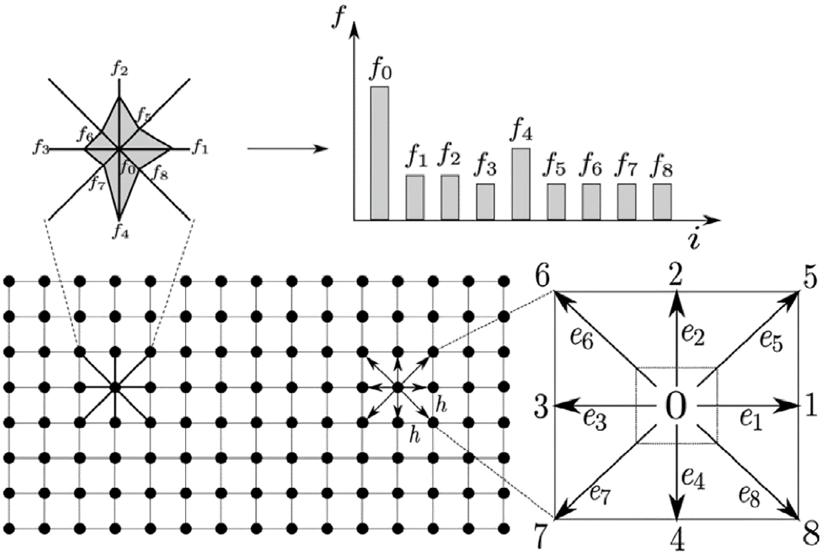


Figure 5.1: Visualization showing the basic idea of the distribution function f_i. The microscopic particle will remain in the same node with highest probability. The second most probable direction of its movement is in direction $\vec{e}_4$ [Maq17].

In short, when using Navier-Stokes equations we are interested in calculating the macroscopic continuous values such as density and velocity directly. When using LBM we calculate these values from the microscopic properties of the fluid (its molecules and their chaotic movement). This is done by establishing links between the discrete properties and their continuous counterparts [Maq17]. For the description of these properties we use the DdQq notation, where d is the number of dimensions (in our case 2 and 3) and q is the number of possible streaming directions. The most common model for 2D is D2Q9 where the streaming of particles is done in 9 directions including zero vector $\vec{e}_0$ as shown in Figure 5.2. For 3D, the most common models are D3Q15, D3Q19 and D3Q27. As described in [WBSP18, SN10], the D3Q19 keeps the computational costs low while maintaining an isotropic lattice, hence why it was chosen in this thesis.

The algorithm can be decomposed into multiple steps which are not dependent on the dimensionality. Let us first look at the two main steps that are essential for LBM: the streaming step and the collision step.

5.1 Streaming Step

The idea of the streaming step is very simple. We propagate the particle densities in the streaming directions as described by the DdQq notation. For each lattice node, we compute the updated distribution function using the values from previous frame as you can see in Figure 5.2 where magnitudes of vectors $\vec{f}_i$ are values of the distribution function in the streaming directions. The same principle as in Figure 5.2 applies in 3D, where the streaming is done in 19 directions instead.

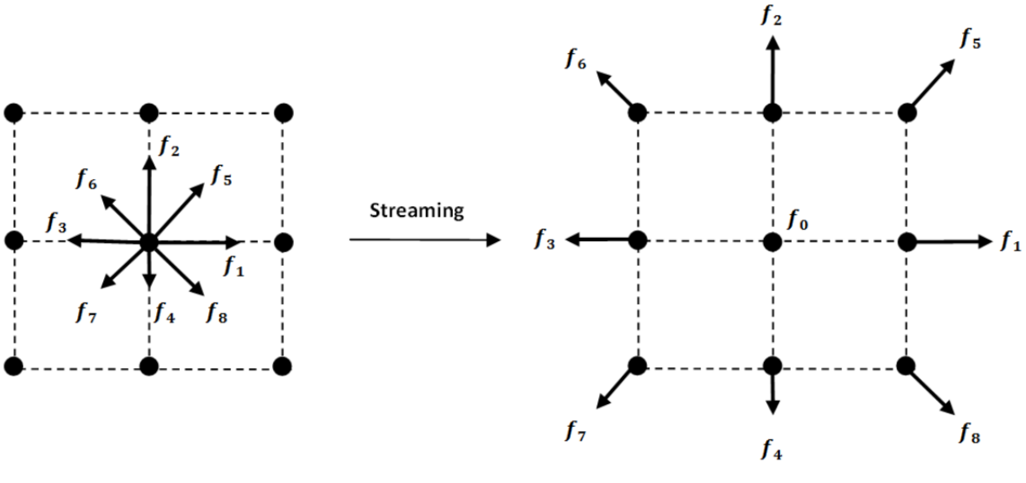


Figure 5.2: Streaming step for D2Q9 configuration [BM11].

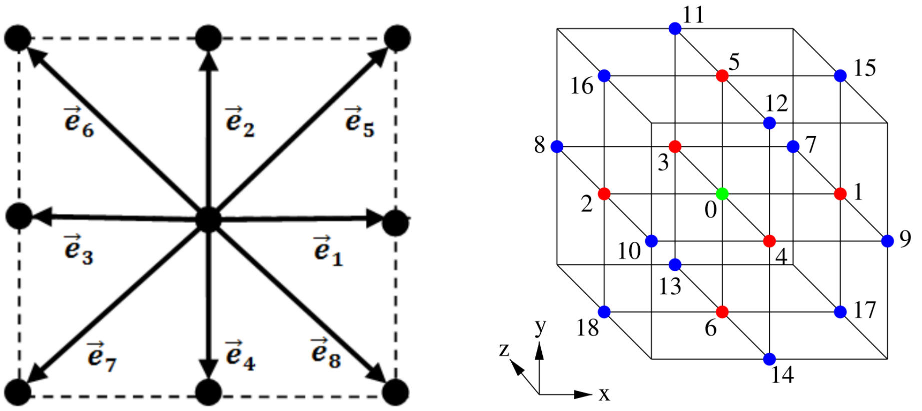
Let $\vec{u}$ be the macroscopic velocity of the particle. Vector $\vec{u}$ is simply a d dimensional velocity vector (so either 2D or 3D). As described in [BM11], the microscopic velocity vector $\vec{e}_i$ for the D2Q9 model is defined as

$$\vec{e}_i = \begin{cases} (0,0) & i=0 \\ (1,0),(0,1),(-1,0),(0,-1) & i=1-4 \\ (1,1),(-1,1),(-1,-1),(1,-1) & i=5-8 \end{cases} \tag{5.1}$$

For the 3D case we have chosen to use the third ordering proposed by Woodgate et al. [WBSP18] that is described by

$$\vec{e}_i = \begin{cases} (0,0,0) & i=0 \\ (\pm1,0,0),(0,0,\pm1),(0,\pm1,0) & i=1-6 \\ (\pm1,0,\pm1) & i=7-10 \\ (0,\pm1,\pm1) & i=11-14 \\ (\pm1,\pm1,0) & i=15-18 \end{cases} \tag{5.2}$$

These orderings can be seen in Figure 5.3



(a) D2Q9 ordering [BM11].

(b) D3Q19 ordering. Third possible ordering as proposed in [WBSP18].

Figure 5.3: Selected DdQq orderings for our implementation.

The distribution function that we compute during the streaming step is denoted f_i^* and is important later on when the molecular particles collide during the collision step.

5.2 Collision Step

The second main step is called the collision step. Collisions between the microscopic particles are simulated, thus obtaining the particle distribution function in the next simulation frame. First, we need to compute the macroscopic density ρ which is given as the sum of distribution function values in the given node $\vec{x}$ as

$$\rho(\vec{x},t) = \sum_{i=0}^{q} f_i(\vec{x},t) \tag{5.3}$$

Provided with ρ, we can compute the macroscopic velocity $\vec{u}$ as

$$\vec{u}(\vec{x}, t) = \frac{1}{\rho} \sum_{i=0}^{q} c f_i(\vec{x}, t) \vec{e}_i \tag{5.4}$$

where c is the lattice velocity. Now that we have the macroscopic velocity $\vec{u}$ of the examined node, we need to compute the equilibrium and update the distribution function f_i. The Bhatnagar-Gross-Krook (BGK) collision operator is used to find the equilibrium distribution $f_i^{eq}(\vec{x}, t)$ for single phase flows (e.g. water, air, steam) as follows

$$f_i^{eq}(\vec{x}, t) = w_i \rho + \rho s_i(\vec{u}(\vec{x}, t)) \tag{5.5}$$

where $s_i(\vec{u})$ is defined as

$$s_i(\vec{u}) = w_i \left[3 \frac{\vec{e}_i \cdot \vec{u}}{c} + \frac{9}{2} \frac{(\vec{e}_i \cdot \vec{u})^2}{c^2} - \frac{3}{2} \frac{\vec{u}^2}{c^2} \right] \tag{5.6}$$

and w_i are the weights (in 2D):

$$w_i = \begin{cases} 4/9 & i = 0 \\ 1/9 & i = 1 - 4 \\ 1/36 & i = 5 - 8 \end{cases} \tag{5.7}$$

and in 3D:

$$w_i = \begin{cases} 1/3 & i = 0 \\ 1/18 & i = 1 - 6 \\ 1/36 & i = 7 - 18 \end{cases} \tag{5.8}$$

The equilibrium function can be also extended to third order scheme as follows [DW16]

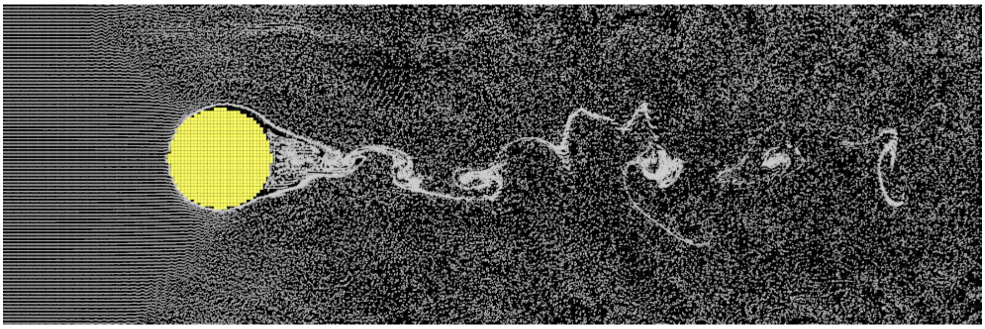
$$s_i(\vec{u}) = w_i \left[3 \frac{\vec{e}_i \cdot \vec{u}}{c} + \frac{9}{2} \frac{(\vec{e}_i \cdot \vec{u})^2}{c^2} - \frac{3}{2} \frac{\vec{u}^2}{c^2} + \frac{\vec{e}_i \cdot \vec{u}}{6c^2} \left(\frac{(9\vec{e} \cdot \vec{u})^2}{c^4} - \frac{3\vec{u}^2}{c^2} \right) \right] \tag{5.9}$$

This results in more stable simulation at the cost of higher computational complexity.

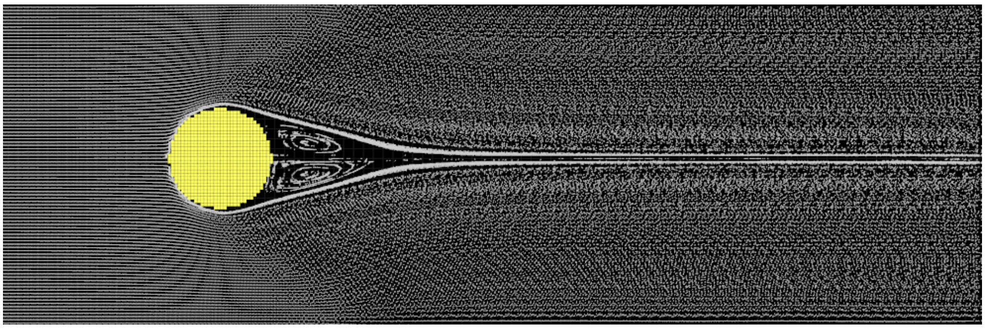
Let f_i^* be the distribution function of node $\vec{x}$ that was already processed in the streaming step as mentioned earlier. The final distribution function for regular lattice node after the collision step is given by

$$f_i = f_i^* - \frac{1}{\tau}(f_i^* - f_i^{eq}) \tag{5.10}$$

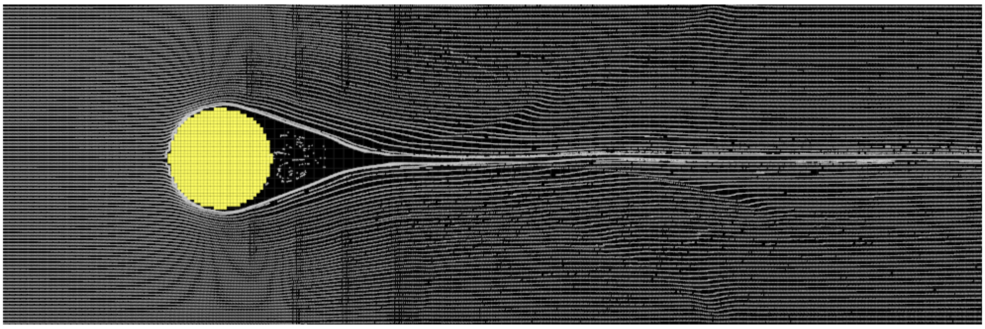
where τ is the relaxation time which is directly related to the viscosity of the fluid. When $\tau \to 1/2$, numerical instability may arise [BM11]. Simulation results with different τ values are shown in Figure 5.4.



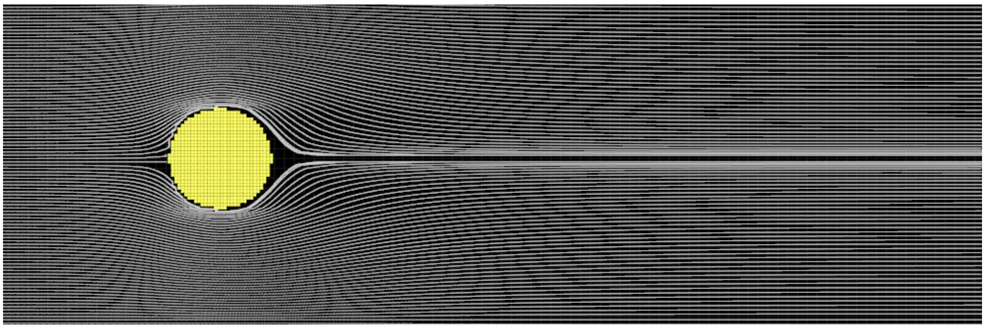
(a) $\tau = 0.55$



(b) $\tau = 0.7$



(c) $\tau = 1.0$



(d) $\tau = 10.0$

Figure 5.4: Flow around circular obstacle with different τ values.

5.3 Inlets

To create a flow in the simulation space, we need to define an initial macroscopic velocity in selected lattice nodes. We call these nodes inlets. This step is very similar to the collision step, only difference is that we do not compute the macroscopic density and velocity. Macroscopic density is set to 1.0 and velocity is user-defined. Equilibrium is then established and the same computation as in collision step is required.

5.4 Obstacles and Boundaries

The problem of obstacles in the scene can be solved simply by using a bounce back model as described in [SN10, BM11]. The idea is that we reverse the distribution function values (left value becomes the right value, etc.) after the streaming step. This approach generates very rough surfaces where the fluid particles close to surface get stuck. As described by Schreiber [SN10], advantage of this approach lies in its simplicity. We do not need any knowledge about the boundary geometry such as its normals. Furthermore, the full bounce back model is well suited for parallelized GPU implementation since there are no memory accesses beyond the distribution function values of the obstacle node. You can see in Figure 5.5 a similar approach where only the incoming distribution vectors are reversed.

Bounce back models come in other variants as well. As an example, Bao [BM11] describes a mid-grid bounce back model that introduces fictitious nodes which are placed between obstacle nodes and the fluid.

Second option to bounce back boundaries are slip boundaries. These are used to represent hydrophobic and slippery surfaces like lotus leafs for example[SN10]. Since both mid-grid bounce back model and slip boundaries require additional information and aren't as easily parallelizable, they are out of scope of this work.

The obstacles are represented by a simple heightmap in 3D. To determine whether a lattice node is an obstacle we check its height and compare it to the heightmap. If the node lies below the heightmap, it is considered as an obstacle.

5.5 Particle Advection

To transport our particles we need to move them according to the macroscopic velocities that are described in the lattice. Here, we find out the coordinate position of each particle and calculate bilinear or trilinear interpolation of the velocities that surround the particle in 2D and 3D, respectively. In other words, in 2D, we find four adjacent lattice nodes that surround the particle and interpolate their macroscopic velocities. In 3D, we find eight adjacent nodes and do the same. The final interpolated velocity is then added to the position of the particle.

Because we are simulating a part of Earth's atmosphere, boundaries of the simulation area cannot act as walls or as slip boundaries. Here we have multiple options what

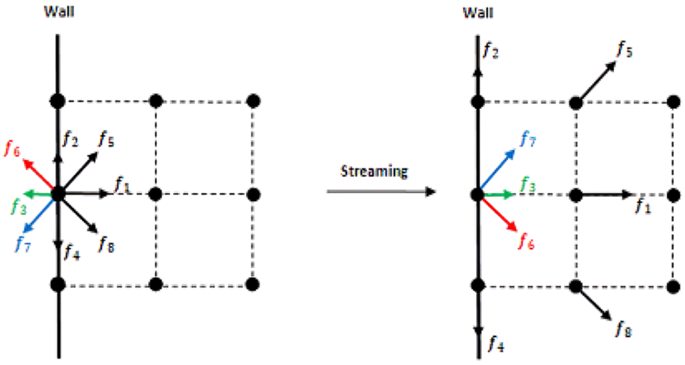

Wall
f_6
f_2
f_5
f_1
f_3
f_7
f_4
f_8
Streaming
Wall
f_2
f_5
f_7
f_3
f_1
f_6
f_8
f_4

Figure 5.5: Full bounce back as presented in [BM11].

to do with particles that find themselves beyond the lattice walls. First, we can deactivate these particles and wait for their reactivation by other simulation components. Secondly, the particles can be respawned in an inlet wall of our selection (or any node for that matter) using a uniform random distribution. Lastly, we can cycle the position of particles. This means that we respawn the particle on the opposite side of the lattice as if it were a tile in a repeating pattern. As an example, consider a 2D lattice. If a particle were to leave a 2D simulation area at the top (its y coordinate is larger then height of the area), it would keep its x coordinate value and $y \leftarrow 0$.

5.6 Main Loop

The simulation steps described above can be ordered in many ways and are presented differently in variety of articles. The approach we have decided to implement uses the order of steps as shown in Figure 5.6.

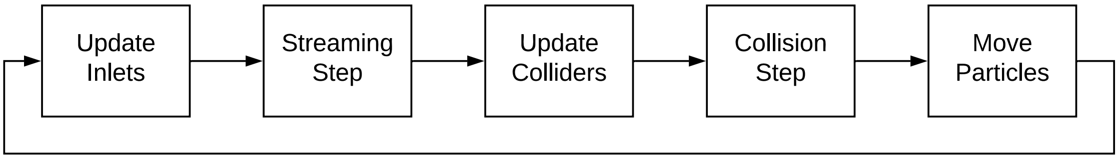

Update Inlets
Streaming Step
Update Colliders
Collision Step
Move Particles

Figure 5.6: The order of simulation steps in our LBM implementation.

6 Cloud Rendering

Before delving into our selected method of cloud rendering let us have a very brief look at basic concepts of cloud radiometry. These concepts are essential in understanding how to approach cloud rendering in general. As mentioned earlier, clouds are composed of a large number of water droplets. These droplets interact with the incident (incoming) light rays/photons by scattering them in many directions or by absorbing their energy. We distinguish between single and multiple scattering. When the light passing through the cloud reaches the viewer after being scattered only once - i.e. after passing through only one water droplet, we describe this as single scattering. On the other hand, when the particle passes through multiple water droplets, we call it multiple scattering as shown in Figure 6.1 [DIYN17]. Single scattering occurs in media that are composed of very small particles or that are very transparent, we denote such media as optically thin. Multiple scattering occurs in media such as clouds where the light almost always exits its volume after multiple scattering events. These media are denoted as optically thick [HL01].

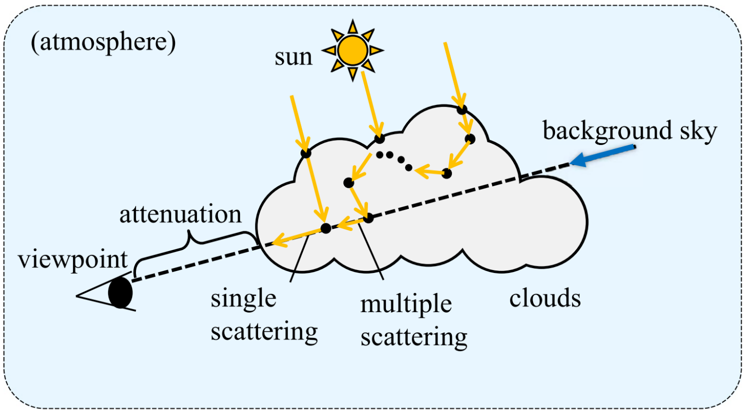


Figure 6.1: Diagram depicting single and multiple scattering events of light inside a cloud [DIYN17].

6.1 Algorithm

Wide selection of methods can be used for rendering clouds as described in Chapter 3. Since we use a particle system to represent the cloud's density distribution, we can either draw the particles at hand, or we can voxelize the particle data into some discrete grid. Because we want to avoid additional costs of converting our cloud representation, a particle based technique was selected as an optimal solution.

Our initial choice was an approach proposed by Harris in his dissertation thesis from 2001 [HL01]. This approach was later deemed not suitable since it assumes static scenes that require a preprocessing step. In it, each particle is drawn from the light's point of view individually to a single framebuffer. Its color is then read back from the framebuffer and saved to an auxiliary array. Using the precomputed array of particle colors, the scene can be rendered at real-time provided the particles do not change their positions. Additionally, Harris uses impostors to alleviate the high rate of pixel overdraw that occurs when a camera is moved too close to a group of particles.

In the end, we have decided to reimplement the particle volume rendering approach as proposed by Green [Gre08]. It is a method that produces very nice results in real-time and does not require any additional data than particle positions and auxiliary framebuffers. The idea is based on so-called *half-angle* slice rendering. A half-angle between view vector $\vec{v}$ and light vector $\vec{l}$ is computed. In cases where the light and view are roughly facing the same way ($\cos(\theta) < 0$ where $\theta = \angle(\vec{v}, \vec{l})$), the half-angle vector is simply computed as their normalized sum. In cases where the light vector is opposite to the view vector, the half-angle vector is computed from the light vector and the inverse of view vector as shown in Figure 6.2.

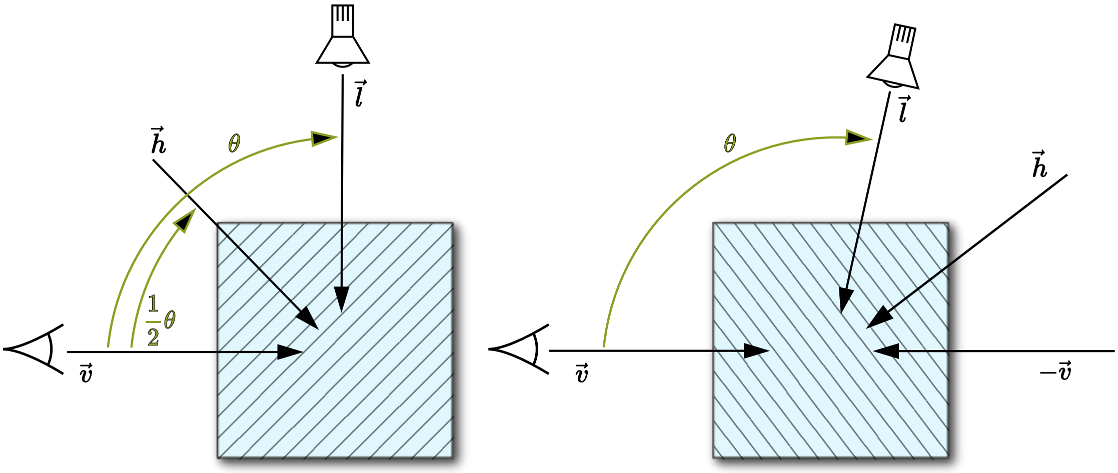


Figure 6.2: Computation of the half-angle vector based on the mutual eye (camera) and light positions [Gre08].

6.2 Particle Rendering

Particles are sorted based on their projection onto the half-angle vector defined axis by using a simple dot product. As stated before, this method does not require any sort of conversion of the particles to a discrete grid of voxels. The only thing we need are the particle positions in world space and an ability to draw them as point sprites in the terminology of OpenGL or as a set of billboards in more general terms.

The rendering is done in batches called *slices*. The smaller each slice is, the higher precision results we obtain at the cost of more draw calls and render target changes. Green suggests that 32 to 128 slices is a good trade-off between visual quality and performance.

First, a single slice is drawn to a framebuffer that holds the intensity of light reaching each particle. This is done using the projection of the light source for which the lighting is being computed. In the case of the sun, we use an orthographic projection the same way we would when computing shadow maps. After the batch has been rendered to the light buffer, we draw the particles from the camera's point of view to an auxiliary framebuffer using the screen projection. During this stage, we sample the light buffer from which we determine how lit each particle fragment is. The algorithm iterates over all slices until all particles are drawn.

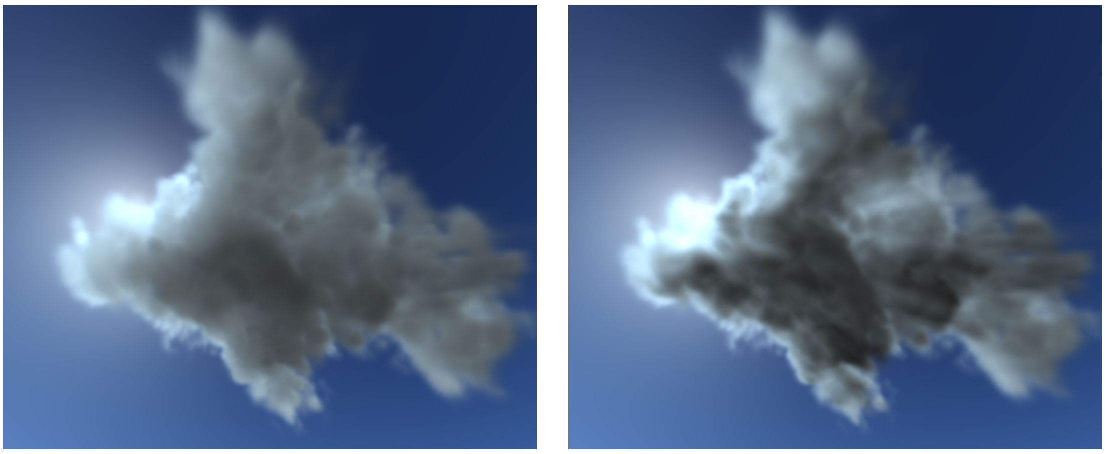
The alpha blending of particle rendering depends on the half-angle vector computation (i.e. on the mutual position of camera and light). The particles are in all cases drawn from front-to-back from the light's point of view. This means that if $\theta \in [0, 90] \rightarrow \cos\theta > 0$, the particles are drawn front-to-back from the camera's point of view. In this case, we say that the view is *inverted* and the blending function is set to `(GL_ONE_MINUS_DST_ALPHA, GL_ONE)`. On the other hand, if the view is not inverted, we draw the particles back-to-front from the camera's point of view and the common blending of `(GL_ONE, GL_ONE_MINUS_SRC_ALPHA)` is used. In both cases, we premultiply the RGB values of the fragment with the alpha channel value in the fragment shader.

6.3 Cast Shadows

Since we use a framebuffer for accumulating particle densities from the light's point of view, we basically create an inverted shadow map. This can be used to draw shadows that are cast by the clouds onto terrain and other objects by simply taking its inverted values `(1.0 - textureIntensity)`.

6.4 Light Texture Blurring

As suggested in the article by Green [Gre08], blurring the light texture after drawing each slice approximates multiple scattering throughout the medium. This is particularly useful in our case since volume of clouds scatters the light in all directions to some degree. Of course, the predominant scattering direction is forward as described by Harris [Har03]. Nonetheless, blurring the light texture is an important improvement of the method which gives us much more realistic results as shown in Figure 6.3.



(a) Light texture blurred with 4 diagonal samples.

(b) No blur applied.

Figure 6.3: Comparison of cloud visualization with and without blur.

6.5 Cloud Occlusion

The cloud rendering described in previous the sections should take place in an auxiliary framebuffer. For clouds to be occluded by other objects such as mountain peaks or trees when viewed from ground, depth map containing depths of these opaque objects must be attached to the auxiliary framebuffer. After the particle rendering is finished, the final image is composited into our scene image.

6.6 Phase Function

One large factor of cloud visualization is the directionality of the scattering light. Since we use an approach that is not physically-based, we lack some of the properties we would like to see in our visualization. Thankfully, directionality of scattering can be easily added to the current model with a little trick that is not necessarily physically correct, but provides us with good results for little cost. For description of these properties we use a *phase function*. Phase function is a function of direction that determines how much light from incident direction $\vec{\omega}$ is scattered into the exitant direction $\vec{\omega}'$ [Har03].

The phase function P is normalized and only depends on the phase angle ϕ where $\cos\phi = \vec{\omega} \cdot \vec{\omega}'$. The phase function is described as:

$$\int_{4\pi} P(\vec{\omega}, \vec{\omega}')d\vec{\omega}' = 1 \tag{6.1}$$

The phase function is reciprocal: $P(\vec{\omega}, \vec{\omega}') = P(\vec{\omega}', \vec{\omega})$ at the same point. The mean cosine g of the scattering angle is defined as:

$$g = \int_{4\pi} P(\vec{\omega}, \vec{\omega}')(\vec{\omega}, \vec{\omega}')d\vec{\omega}' \tag{6.2}$$

If the mean cosine is 0, the scattering is isotropic. If g is negative, backward scattering dominates, and if g is positive, the scattering points mainly in forward direction [Pre03]. There exists a vast array of phase functions where each describes scattering in particles of different sizes and other parameters. Let us denote the phase functions based on the scattering angle ϕ as shown in Figure 6.4 as $P(\vec{\omega}, \vec{\omega}') = P(\phi)$.

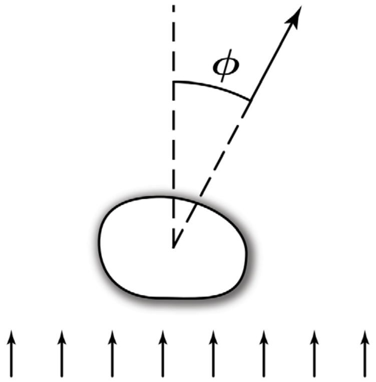


Figure 6.4: The phase function scattering angle ϕ is the angle between the incident (bottom arrows) and scattered (top right arrow) light directions [Har03].

6.6.1 Isotropic Phase Function

The simplest phase function where the light is scattered in random directions with equal probability is called isotropic. It is defined as:

$$P(\phi) = \frac{1}{4\pi} \tag{6.3}$$

6.6.2 Rayleigh Phase Function

Scattering inside very small particles such as those found in clear air can be approximated using Rayleigh phase function. It is described as:

$$P_{Ray}(\phi) = \frac{3}{4}\frac{(1+\cos^2\phi)}{\lambda^4} \tag{6.4}$$

For this phase function to be valid, the size of the particle must be smaller than the wavelength λ of the light passing through it. Harris uses Rayleigh phase function in his cloud rendering algorithm due to its low computational cost. It is implemented in our framework but since it is not valid for particles of water vapor due to their size, the results are not desirable. Let us therefore look at more accurate approximations for larger particles such as water droplets.

6.6.3 Henyey-Greenstein Phase Function

The Henyey-Greenstein phase function is very useful in approximating scattering in water, clouds, biological tissues and many other natural materials [Pre03]. It simplifies Gustav Mie's theory of light scattering by larger particles. The phase function is described as:

$$P_{HG}(\phi, g) = \frac{1}{4\pi} \frac{1 - g^2}{(1 - 2g\cos\phi + g^2)^{3/2}} \tag{6.5}$$

Here, g is the symmetry parameter that controls the scattering. Positive values of g indicate that incident light will be scattered in forward direction, while for negative g the light will be scattered in backward direction. For $g = 0$ we get isotropic scattering as described previously. To see a visual comparison for different positive values of g, please see Figure 6.6.

In Figure 6.5 you can see comparison of Rayleigh, Henyey-Greenstein and Mie phase functions. The graph shows the lack of some features that are not captured by Henyey-Greenstein approximation. These features, such as glory and fogbow which are optical phenomena similar to rainbows and halos, are mainly present in the opposite direction (backward scattering).

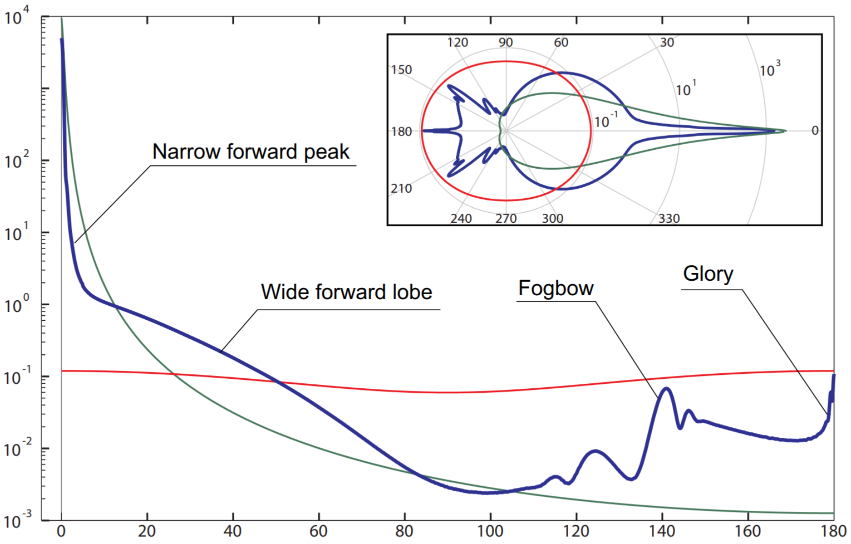


Figure 6.5: Logarithmic plots (inset: polar log plots) of Rayleigh (red), Henyey-Greenstein with $g = 0.99$ (green) and Mie (blue) phase functions. The graph depicts problems and missing features of Henyey-Greenstein phase function when compared to Mie phase function [BNM$^+$08].

6.6.4 Double Henyey-Greenstein Phase Function

The main drawback of Henyey-Greenstein phase function is that it can only capture scattering events in one direction (forward or backward depending on g). To alleviate this issue, Kattawar [Kat75] shows an extension of the phase function using two symmetry operators g_1 and g_2 that captures the front and backward facing lobes of the scattering distribution. It is simply described as:

$$P_{DHG}(\phi) = (1-f)P_{HG}(\phi, g_1) + fP_{HG}(\phi, g_2) \tag{6.6}$$

where $g_1 > 0$ determines intensity of forward scattering and $g_2 < 0$ backward scattering and f is the interpolation parameter.

6.6.5 Schlick Phase Function

While Henyey-Greenstein phase function is a good approximation of Mie scattering, it is still computationally expensive due to the usage of exponentiation (3/2) in fragment shader. Schlick proposes a simpler expression that has a less described shape than the Henyey-Greenstein phase function. The expression is described as:

$$P_{Sch}(\phi, k) = \frac{1}{4\pi} \frac{1-k^2}{(1+k\cos\phi)^2} \tag{6.7}$$

where k is a parameter similar to the asymmetry parameter g and it holds that $-1 \leq k \leq 1$. In our shader implementation, we use an approximation of k that is used in the Frostbite engine by Electronic Arts [Hil16] as

$$k \approx 0.55g^3 - 1.55g \tag{6.8}$$

Note that we use an inverse value as opposed to the suggested approximation in Frostbite since we adhere to the rule that positive g scatters the light in forward direction. The exponents $(1 + k\cos\phi)^2$ can be simply substituted with multiplications in the shader while we cannot get rid of the square root term in the Henyey-Greenstein equation[1].

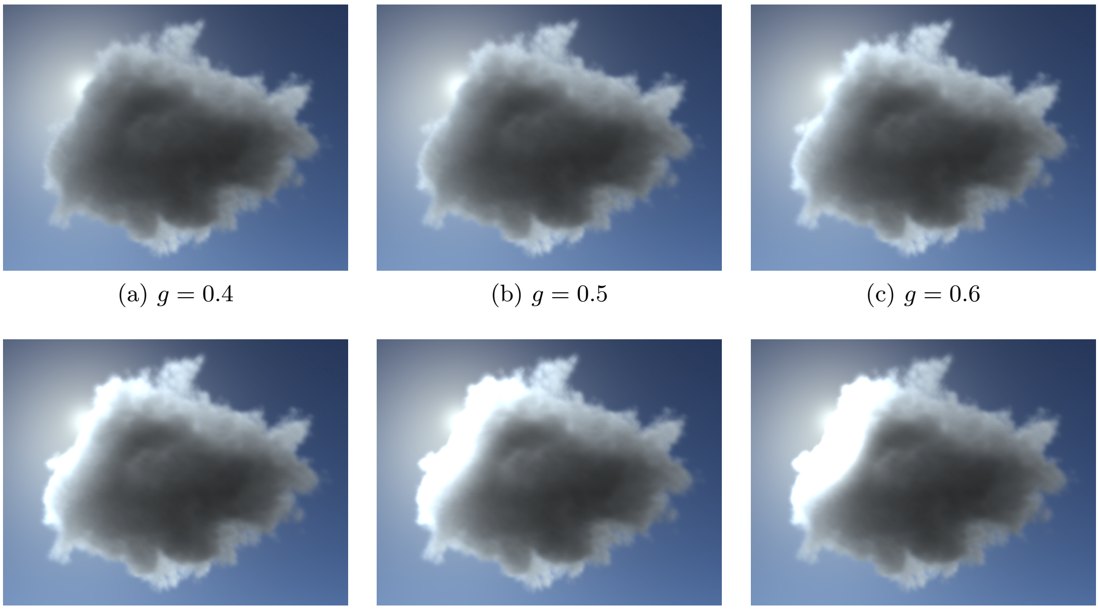
6.6.6 Cornette-Shanks Phase Function

Lastly, a modification of Henyey-Greenstein phase function was presented by Cornette and Shanks [CS92]. The modification uses a more physically realistic approximation that is, as stated by Premože [Pre03], more suitable for clouds. It is defined as:

$$P_{CS}(\phi, g) = \frac{1}{4\pi} \frac{3}{2} \frac{(1-g^2)}{(2+g^2)} \frac{1+\cos^2\phi}{(1+g^2-2g\cos\phi)^{3/2}} \tag{6.9}$$

---

[1] Note that we have observed a difference of approximately 0.5ms when switching between these two phase functions for 1 million particles on our desktop computer (see Table 9.1 for specifications).

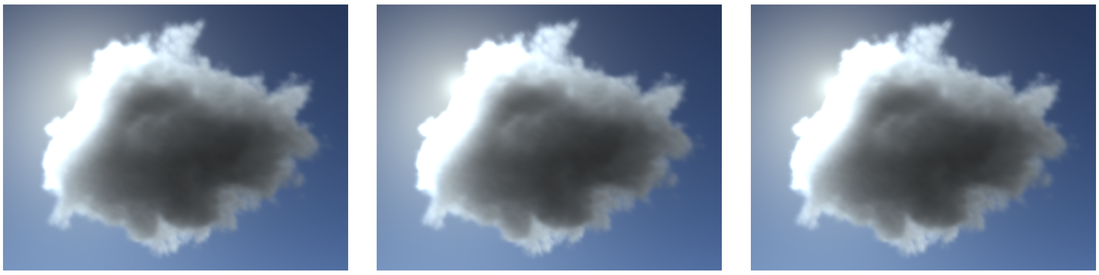


(a) $g = 0.4$ (b) $g = 0.5$ (c) $g = 0.6$

(d) $g = 0.7$ (e) $g = 0.8$ (f) $g = 0.9$

Figure 6.6: Comparison of different values of g for Henyey-Greenstein phase function.

We compare the Henyey-Greenstein, Schlick and Cornette-Shanks phase functions that were integrated into our framework in Figure 6.7. Notice that the results show no notable differences among the functions.



(a) Henyey-Greenstein (b) Schlick (c) Cornette-Shanks

Figure 6.7: Comparison of different phase functions with $g = 0.8$.

7 Solution Proposal

In this chapter we would like to look at what our solutions are in integrating the previously described systems and algorithms into our framework. Main goal of this thesis is an extension of Duarte's method with a wind flow field generated using LBM. Since Duarte's method works with simple particles where we only need to know their positions, velocities, and convective temperatures (i.e. which profile they belong to), coupling of these two methods isn't problematic. Another goal of this thesis is parallelization of Duarte's method on the GPU which is necessary for large amounts of particles. The same holds for LBM where parallelization is necessary for real-time simulations. Besides significant speed ups of both methods, their parallelization also allows us to store the data only on GPU without ever needing to send it back to the CPU after initialization of all our systems. Simply put, our simulator should run purely on GPU.

We propose a simple architecture for our application that is centered around the `ParticleSystem` class. The `ParticleSystem` class holds all data pertaining to the particles themselves. These are then accessed by individual simulators of the application as shown in Figure 7.1.

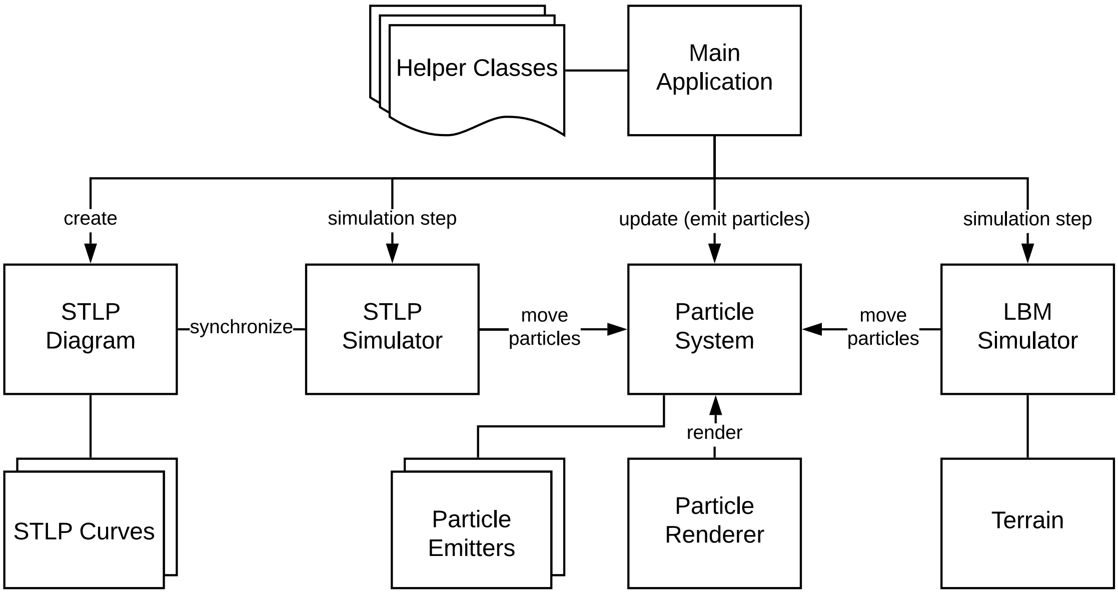

Helper Classes
Main Application
create
simulation step
update (emit particles)
simulation step
STLP Diagram
synchronize
STLP Simulator
move particles
Particle System
move particles
LBM Simulator
render
STLP Curves
Particle Emitters
Particle Renderer
Terrain

Figure 7.1: Proposed simplified architecture of the application.

The simulators: `STLPSimulator` and `LBMSimulator` are independent entities when it comes to their usage. In other words, each simulator can advect the particles by itself and can be enabled or disabled on the run without affecting the other. Let us look at how each should be incorporated into our framework.

7.1 SkewT/LogP Simulation

The SkewT/LogP (STLP) simulation runs on the GPU since the proposed method by Duarte is easily parallelizable. The system initially loads the STLP diagram and then parses it to the simulator for use. The diagram is loaded on the CPU and its curves are uploaded to vertex buffer objects (VBOs) for rendering. The `STLPSimulator` class loads only the necessary curves for simulation to CUDA memory. The system is designed to be able to handle diagrams of variable sizes/ground altitudes. It does not presume length of the curves and gives its users free hand at modifying curve smoothness. The system should also support loading and editing diagrams at runtime. The simulator must therefore update the data stored on GPU when the diagram is changed in any way.

As described in previous chapters, Duarte's method includes a naive wind simulation that uses the sounding data for particle advection. Since we want to conserve memory and the wind is simulated by LBM, we can omit particle velocity vectors $\vec{v}$ from our simulation and only use a single float to describe its vertical velocity v_y. This is possible because LBM does not need information about a particle's velocity in previous simulation step. We would like to note that Duarte's naive wind from his cumulus cloud simulation was implemented on the CPU but was later deprecated in our system when migrating to GPU simulation only. Moreover, we have also tried applying wind data obtained from soundings as an inlet velocity array for LBM. This however exhibited erratic behavior that in most cases resulted in unstable LBM simulation and was deprecated in our application.

We also omit Duarte's terrain influence in his orographic simulation. The terrain influence by Duarte tests all terrain mesh triangles with a given velocity vector when searching for intersection point. This is very costly without any acceleration data structures such as grids, octress, kd-trees, or others. This problem is fortunately not present when the terrain is defined by a heightmap. Nonetheless, for collisions/flow around obstacles we use LBM which solves flow around the terrain implicitly.

7.2 Lattice Boltzmann Method Simulation

Thankfully, LBM is easily decoupled from other systems and can therefore stand on its own. Note that Duarte's system is an inherently lagrangian system, meaning that we label each particle and track it anywhere in world space. This can be seen as one of its strengths. On the other hand, LBM is an eulerian method. This means that we track fixed points in the space (the lattice) and their properties. If we want to simulate our particles using the LBM, we need to keep them inside the lattice's simulation area. Now let us consider that we want a vast terrain that would require large and computationally expensive simulation area. For this reason, we would like the user to have control over it. Our proposed solution is to use a positionable LBM simulation area. Furthermore, we propose that said area can also be scaled. The scaling determines lattice cell size in world units.

This however poses problems for the LBM simulation. Due to time constraints, parametrized LBM simulation was not implemented in the framework. This means that by scaling the simulation area, one also scales the resulting velocities that move the particles. This is one area of our system that would greatly benefit from future improvements. As a naive solution, we add a velocity multiplier that is applied during the advection of particles. Using this, users are able to speed up or slow down wind speed. This however does not change properties of the lattice itself and therefore may look unnatural for multipliers that are too large or too small. The actual lattice resolution, i.e. the number of cells on each axis, does not have this problem and should also be configurable using a configuration file.

7.3 Viewports

The application should also provide two viewports. One is the general 3D viewport where users can look at the generated clouds, the terrain and much more. The other is a 2D STLP diagram viewport where the user should be able to inspect and edit the loaded STLP diagram. The framework should also support drawing the diagram in the 3D viewport as an overlay texture. The proposed appearance of the 3D viewport without UI is shown in Figure 7.2 where the LBM simulation area is visualized.

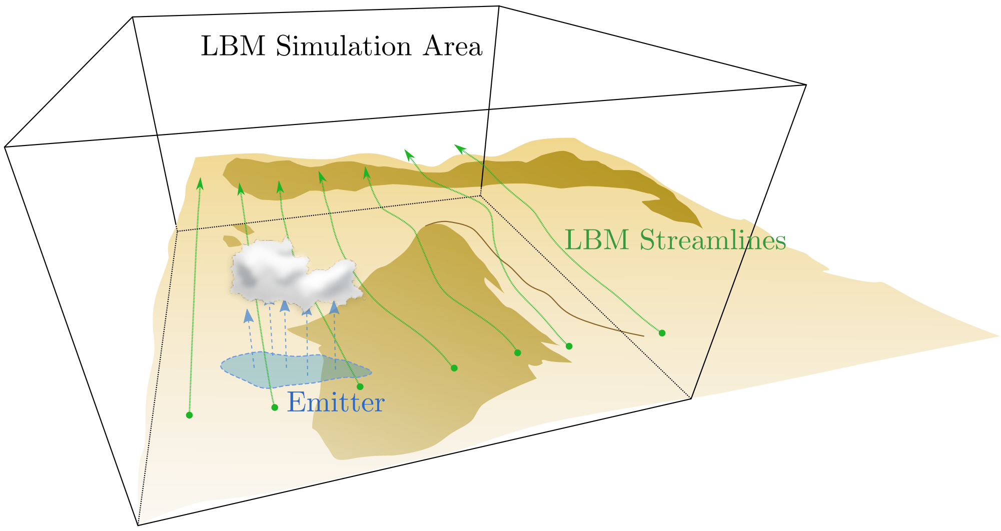


Figure 7.2: Proposed 3D viewport of our application.

7.4 Terrain

Since we use a terrain as an obstacle in LBM, LBM needs to have access to its data. In the proposed system, everytime the terrain is generated or edited, the flattened normalized height data is uploaded to the LBM simulator for use.

Furthermore, the framework should be able to generate random terrains using perlin noise or load height data from grayscale textures. Both these features should be available in the user interface of our application.

Terrain rendering is also considered. Multiple materials can be used when rendering the terrain. For determining how much each material contributes to certain terrain vertex we propose using simple RGBA textures that give us option to mix materials without visible borders between them.

7.5 Cloud Rendering and Sky

Our framework supports cloud rendering using the described half-angle slicing method. One contribution of our work is integration of anisotropic scattering approximation to this method by using phase functions to increase intensity of selected particles. Additionally, to improve visual fidelity of the framework, sky model proposed by Hošek and Wilkie [HW12] was added to the system. We sample the sky at sun position to tint the directional light that is used as the sun.

7.6 Rendering

Since we want to present our results in gorgeous renders, we need to render the terrain, materials and any other objects using modern shaders. For this, we have implemented a simple physically-based rendering (PBR) shader system based on the tutorials by Vries [dVa]. Furthermore, our system uses advanced shadow mapping technique called exponential variance shadow maps (EVSM) which allows us to create soft and high-quality shadows by blurring the depth map texture and by using mipmapping and anisotropic filtering on said texture. EVSM is further explained in Appendix B.

7.7 Debugging Tools and User Control

The proposed framework needs to be versatile and give users a free hand at customizing its individual systems. For this we add an extensive user interface that provides users with options to modify parameters of said systems. If any parameters are not modifiable at runtime, users are given an option to configure the application using a `config.ini` file or command line arguments.

7.8 World Unit Size

One major problem that we encountered when creating the application was the world unit size. Due to the fact that the backbone of the whole application was initially a pure LBM simulator, other systems adhered to the cell size of the lattice. In the final application, the roles were reversed. Standard unit size of 1 meter is used throughout the framework and the LBM simulator converts particle coordinates from world space coordinate system to its own coordinate system and back when necessary.

8 Implementation

Let us now look at selected implementation details of our framework. Since the created framework is quite sizeable, only the most important and interesting topics and code snippets will be presented. For a reader that wants to delve into our system head-on we also provide Doxygen generated documentation for our framework[1].

8.1 Particle System and Memory Management

One of the main goals of this thesis is a proper utilization of the GPU and its capabilities where possible. Since we are simulating clouds using vast amounts of particles, it is crucial that we keep the number of data transfers between the GPU and CPU to a minimum. At the core of this ideology is the `ParticleSystem` class[2] that holds pointers to the GPU memory where we store all necessary attributes needed for the simulation. Unfortunately, we cannot represent individual particles by a simple class or struct since the data is divided into OpenGL managed memory and CUDA managed memory. OpenGL requires particle positions and profile indices (i.e. index of a convective temperature (T_c) profile they belong to). These are stored in regular vertex buffer objects (VBOs). Furthermore, element buffer object (EBO) is used for indexation of the particles whose main purpose lies in their sorting. Instead of sorting multiple individual buffers (and CUDA arrays), we only sort particle indices which are then used in `glDrawElements` draw call instead of a simple `glDrawArrays` draw call. Both VBOs and the EBO are registered and mapped by CUDA for use in kernels during the simulation.

Besides these buffers, we also need to store particle velocities, more specifically, their vertical velocity used in STLP. This information is only necessary in CUDA kernels and is therefore stored in global device (GPU) memory. Lastly, we want the users to have control over the displayed amount of particles at any moment. This is solved by preloading a maximum amount of particles as defined in a configuration file and drawing/simulating only those that are active. All active particles are the leftmost particles in the arrays described above. At this moment, individual particles at random memory location cannot be deactivated apart from the last one (rightmost one). Activation and deactivation of the last active particle (or batches of particles) is used by emitters as described in Section 8.8.

For easy cloud prototyping we provide a very simple save & load feature that writes particle positions and their profile indices to binary files. By using binary file streams the application loads and saves a million particles in approximately 25ms and 45ms on our test desktop machine (see Table 9.1), respectively. In the loading process, number of saved particles is checked against maximum available memory, and in case of overflow, only the permitted amount is loaded. Particles are then uploaded either using `glNamedBufferSubData` or `glNamedBufferData` to GPU based on whether they match the maximum current capacity. For saving the particles, the GPU buffers must be first mapped to CPU by using `glMapNamedBuffer` function before its content is written to the save file.

---

[1] available at `https://www.martincap.io/ProjectFuji/doc/`

[2] defined in `ParticleSystem.h` header file

8.2 SkewT/LogP Diagrams

Simple SkewT/LogP diagram[1] visualization and user interface was implemented as part of our framework. The sounding data is loaded from regular text files using the data layout from `www.twisterdata.com`. The initial sounding data file can be set in `config.ini` file or as a command line argument. Furthermore, all text files present in the predetermined sounding file directory are preloaded. This means that the user can load new sounding data at runtime provided its files are in the correct directory.

One small complication of the diagram visualization lies in the fact that the y axis is reversed in the orthographic projection due to the fact that pressure is displayed from highest to lowest on the positive y axis. This is taken into account in the mapping function that skews the temperature axis and uses logarithm with base 10 on the pressures. The particular implementation of these two-way mapping functions (in the implementation denoted as the normalization process) can be seen in Listing 8.1. Note that we need to provide the y axis value when computing x for given temperature T and vice versa. This means that the y axis value has to be always computed first.

```
float getNormalizedPres(float P) {
    return ((log10f(P) - log10f(MIN_P)) / (log10f(MAX_P) - log10f(MIN_P)));
}
float getNormalizedTemp(float T, float y) {
    return (T - MIN_TEMP) / (MAX_TEMP - MIN_TEMP) + (1.0f - y);
}
float getDenormalizedPres(float y) {
    return powf(10.0f, y * (log10f(MAX_P) - log10f(MIN_P)) + log10f(MIN_P));
}
float getDenormalizedTemp(float x, float y) {
    return (x + y - 1.0f) * (MAX_TEMP - MIN_TEMP) + MIN_TEMP;
}
```

Listing 8.1: Mapping and unmapping functions used for diagram creation.

We gather the corresponding curves into groups and upload them as a single array to a VBO to reduce the necessary number of draw calls when rendering the diagram. This comes at a price of a less scalable system that requires core changes if we were to modify individual curves.

---

[1] defined in `STLPDiagram.h` with additional utility functions from `STLPUtils.h`

8.2.1 Dry Adiabat Creation

Dry adiabats determine the dry lift motion of the particles. Their creation uses a non-iterative approach that generates a curve for constant potential temperature θ. In other words, we can compute any point (T, P) of the dry adiabat for a potential temperature θ without any additional data just by using Equation 4.7. The algorithm for generating a single dry adiabat for given θ in degree Celsius is shown in Listing 8.2.

```
createDryAdiabat(float theta, Curve &curve) {
    const float P0 = soundingData.groundPressure;
    const float k = 0.286f;                     // $R_d/c_{pd}$
    const float thetaK = theta + 273.15f;       // convert to Kelvin
    foreach(P in soundingData) {
        float T = thetaK / pow((P0 / P), k);    // Equation 4.7
        T -= 273.15f;                           // get value in Celsius
        float y = getNormalizedPres(P);         // get normalized y coordinate
        float x = getNormalizedTemp(T, y);      // get normalized x coordinate
        curve.addVertex(x, y);
    }
}
```

Listing 8.2: Function for computing dry adiabat with given θ value.

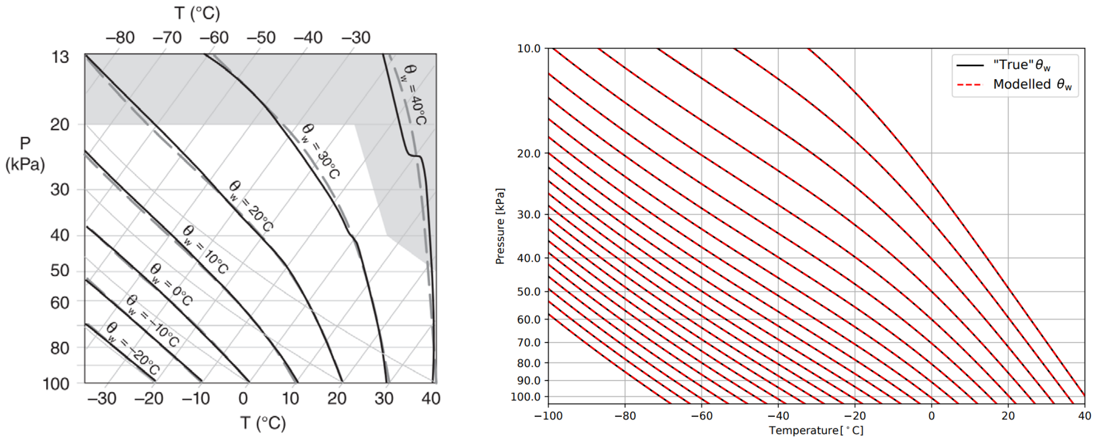
8.2.2 Moist Adiabats Implementation Details

Moist adiabats are the most tricky curves present in the SkewT/LogP diagrams since they need to be created iteratively. We have had some problems with Duarte's equations when it comes to moist adiabat creation and therefore turned to other articles. The issues were later resolved and due to this we provide two approaches as was described in the theoretical part of our text. Interestingly enough, we have also found two non-iterative approaches that try to significantly reduce the computational complexity of generating moist adiabats which we would like to quickly introduce.

Non-Iterative Methods

First of these was proposed by Bakhshaii and Stull [BS13]. The method uses gene-expression programming (GEP) and generates functions that compute T based on the potential temperature θ and pressure P. We have implemented this method and even though the approximations in the area that the algorithm was trained on are correct, we have decided to use the iterative approach in the end. This is mainly because creating the desired moist adiabats on CPU is a preprocessing step that is not significant when it comes to runtime speed. Results of this method can be seen in Figure 8.1a.

Moisseeva and Stull also proposed a non-iterative approach which, according to their paper, provides better results than Bakhshaii's method [MS17]. It uses polynomials to fit the pseudoadiabat curves. The method solves drift issues that can occur in iterative solutions due to numerical integration errors. However, in the end, we have opted to use the iterative approach as to avoid random errors generated by the non-iterative approaches. Results of Moisseeva and Stull's method are presented in Figure 8.1b.



(a) Results of Bakhshaii's method [BS13]. (b) Results of Moisseeva's non-iterative approach [MS17].

Figure 8.1: Moist adiabats obtained by using non-iterative methods. Note that Moisseva's moist adiabats are plotted using an emagram which is another type of thermodynamic diagram.

Iterative Methods

The iterative approach is built on computing individual curve points one after another with small ΔP. To create the moist adiabat, we can iteratively solve for T_2 from given T_1 at P_1 and a $\Delta P = P_2 - P_1$ step (therefore we also know P_2 at each iterative step). We have implemented both the pseudoadiabatic lapse rate described by Duarte (Equation 4.11) and the iterative approach described by Bakhshaii (Equation 4.10). In Listing 8.3 we present an algorithm for creating the moist adiabats including the base of Bakhshaii's method on Line 20.

```
createMoistAdiabat(float theta, float startP, float deltaP, float smallDeltaP) {
    float T = getKelvin(theta);
    float P_Pa;
    float accumulatedP;
    const float smallDeltaP_Pa = smallDeltaP * 100.0f;
    for(float P = startP; P >= MIN_P - smallDeltaP; P -= smallDeltaP) {
        if (accumulatedP >= deltaP || accumulatedP == 0.0f || P <= MIN_P) {
            accumulatedP = 0.0f;                    // reset accumulated pressure
            float y = getNormalizedPres(P);         // get normalized y coordinate
            float x = getNormalizedTemp(T, y);      // get normalized x coordinate
            curve.addVertex(x, y);
        }
        P_Pa = P * 100.0f;   // convert to pascals
        T -= dTdP_moist_degK(T, P_Pa) * smallDelta_Pa;
        accumulatedP += smallDeltaP;
    }
}

// Computes Equation 4.10
float dTdP_moist_degK(float T, float P) {
    float L_v = computeLatentHeatOfVaporisationK(T);    // Equation 4.9
    float w = w_degK(T, P);                             // Equation 4.4
    // $R_d = 287.05307$, $c_{pd} = 1005.7$, EPS $\approx 0.622$
    float res = 1.0f / P;
    res *= (R_d * T + L_v * w);
    res /= (c_pd + (L_v * L_v * w * EPS / (R_d * T * T)));
    return res;
}
```

Listing 8.3: Function for computing moist adiabat with given θ, start P, and delta values.

8.2.3 Intersection of Line Segments in 2D

Intersection of two line segments is an important part of the simulation process when finding necessary attributes such as T_c, CCL and others in the SkewT/LogP diagram. This is why we would like to go a little more into detail of how to find the intersection point of two line segments and if there is any. Furthermore, we would like to very briefly show that not all line segment intersections require such a general approach, especially when we are looking for intersection between isobars and ambient temperature curves which are computed for all particles in each simulation step as was shown in Algorithm 1.

General Approach

First, let us look at the general approach to finding an intersection point between two line segments as shown by Bourke [Bou88] and further explained by Walton [Wal].

Assume two line segments, $\overline{P_1P_2}$ and $\overline{P_3P_4}$. We can define points P_a and P_b on each of these two lines, respectively, as:

$$P_a = P_1 + t_a \cdot (P_2 - P_1) \tag{8.1}$$

$$P_b = P_3 + t_b \cdot (P_4 - P_3) \tag{8.2}$$

where $0 \leq t_{a,b} \leq 1$.

To find an intersection of these line segments, we assume $P_a = P_b$, which gives us

$$P_1 + t_a \cdot (P_2 - P_1) = P_3 + t_b \cdot (P_4 - P_3) \tag{8.3}$$

We need to break this into x and y components. Let $P_i = (x_i, y_i)$ for $i = 1, 2, 3, 4$. We get

$$x_1 + t_a \cdot (x_2 - x_1) = x_3 + t_b \cdot (x_4 - x_3) \tag{8.4}$$

$$y_1 + t_a \cdot (y_2 - y_1) = y_3 + t_b \cdot (y_4 - y_3) \tag{8.5}$$

We can rearrange these equations as follows:

$$(x_3 - x_1) = t_a(x_2 - x_1) - t_b(x_4 - x_3) \tag{8.6}$$

$$(y_3 - y_1) = t_a(y_2 - y_1) - t_b(y_4 - y_3) \tag{8.7}$$

This set of equations can be rewritten into matrix form:

$$\begin{bmatrix} x_2 - x_1 & -(x_4 - x_3) \\ y_2 - y_1 & -(y_4 - y_3) \end{bmatrix} \begin{bmatrix} t_a \\ t_b \end{bmatrix} = \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \end{bmatrix} \tag{8.8}$$

Since t_a and t_b are the only unknown values, we need to rearrange this by using an inverse of the leftmost matrix. Just as a reminder, for any matrix $\mathbf{A} \in \mathbb{R}^{2x2}$, where

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{8.9}$$

its inverse $\mathbf{A}^{-1}$ is defined as:

$$\mathbf{A}^{-1} = \frac{1}{\det \mathbf{A}} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{8.10}$$

where $\det \mathbf{A} = ad - bc$. This results in

$$\begin{bmatrix} t_a \\ t_b \end{bmatrix} = \frac{1}{(x_4 - x_3)(y_2 - y_1) - (x_2 - x_1)(y_4 - y_3)} \begin{bmatrix} -(y_4 - y_3) & x_4 - x_3 \\ -(y_2 - y_1) & x_2 - x_1 \end{bmatrix} \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \end{bmatrix} \tag{8.11}$$

By multiplying, we obtain:

$$t_a = \frac{(x_4 - x_3)(y_3 - y_1) - (x_3 - x_1)(y_4 - y_3)}{(x_4 - x_3)(y_2 - y_1) - (x_2 - x_1)(y_4 - y_3)} \tag{8.12}$$

$$t_b = \frac{(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)}{(x_4 - x_3)(y_2 - y_1) - (x_2 - x_1)(y_4 - y_3)} \tag{8.13}$$

There are 4 possible base results when plugging our two points into these last two equations:

1. **Segments intersect**: If $0 \leq t_a \leq 1$ and $0 \leq t_b \leq 1$ then the line segments intersect. The point of intersection can be found by using either t_a or t_b in Equation 8.1 or Equation 8.2, respectively.

2. **Lines intersect**: If the value of either t_a or t_b falls outside of the range $[0, 1]$, then we are talking about an intersection between the two lines that are defined by the two line segments. More specifically, if one of the two (t_a or t_b) lies in range $[0, 1]$ and the other does not, then we are looking at an intersection of a line segment and a line.

3. **Lines are collinear**: If the denominator in Equation 8.12 or Equation 8.13 is zero, it means that the two lines (and hence the line segments) are collinear. This can be further separated into two cases:

   (a) **Lines do not intersect**: Here, the lines are collinear but not the same.

   (b) **Lines intersect**: Here, the two lines intersect in infinite number of points. This part can also be decomposed since this does not tell us whether the two line segments intersect in infinite amount of points or if only the lines intersect and the line segments do not.

This general approach for finding intersections is only needed on the CPU when creating the diagram and is available in the `Curve.h` header file. Our implementation also offers some additional options such as reversing search order for curve vertices or finding n-th intersection of the two curves.

Isobar Intersection

As the reader can observe, there are multiple cases in our dry and moist lift algorithms where we only need to find an intersection between the ambient temperature curve and a single isobar (which can be interpreted as a line instead of a line segment). Since we assume that the ambient temperature curve C_a is y-monotone (y-strictly-monotone even), we can simplify the intersection search by first looking for a correct segment based on its y coordinate. This initial search can be done either naively using linear search or more cleverly with a binary search. We use a binary search on the GPU for each kernel thread.

Computation of the intersection point with selected segment is also much easier since isobars are horizontal lines defined as $y = k$ for some k, usually $0 \leq k \leq 1$ because we normalize our data (or $100 \leq k \leq 1000$ [hPa] in pressure).

Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be the two end points of the selected line segment of C_a. Let $P_3 = (x_3, y_3)$ be the desired intersection point. We know that $y_3 = k$. Because P_3 is an interpolation of P_1 and P_2, we can express P_3 in terms of P_1 and P_2 as follows:

$$P_3 = tP_1 + (1 - t)P_2 \tag{8.14}$$

When written for each axis separately, we get

$$x_3 = tx_1 + (1 - t)x_2 \tag{8.15}$$

$$y_3 = ty_1 + (1 - t)y_2 \tag{8.16}$$

From Equation 8.16, we can express t as

$$t = \frac{y_3 - y_2}{y_1 - y_2} \tag{8.17}$$

where everything is known as $y_3 = k$. We can therefore plug t into Equation 8.15 to get x_3. We present the algorithm for finding the intersection with isobar in Listing 8.4 with additional out of bounds checking whose purpose is further examined in Section 8.3.

```
vec2 getIntersectionWithIsobar(vec2 *vertices, int n, float y) {
    if (y >= vertices[0].y) { return vertices[0]; }
    if (y <= vertices[n - 1].y) { return vertices[n - 1]; }
    int left = 0;
    int right = n - 1;
    while (left <= right) {
        int curr = (left + right) / 2;
        if (vertices[curr].y > y) {
            left = curr + 1;
        } else if (vertices[curr].y < y) {
            right = curr - 1;
        } else { return vertices[curr]; }
    }
    float t = (y - vertices[left].y) / (vertices[right].y - vertices[left].y);
    float resT = t * vertices[right].x + (1.0f - t) * vertices[left].x;
    return vec2(resT, y);
}
```

Listing 8.4: Isobar intersection using simple binary search and interpolation.

8.2.4 Diagram Customization

Similar to Duarte's approach, we also offer customization of the sounding data curves, particularly the ambient and dew point temperature curves. Position of the individual control points can be adjusted on the appropriate isobar. For this to be possible, when the user enables curve edit mode and uses a mouse button to select a point on the screen, we search for the closest point on the curve. We adjust the point position on the CPU. We leave the actual diagram update, simulation parameter recalculation, and upload to

GPU to the user. We do not want to update these data continually which would be computationally expensive. The application also provides customizable curve colors and any group of curves can be hidden in the menu.

8.2.5 Text Rendering

Text rendering for labeling the diagram was implemented based on the tutorials by Joey de Vries [dVb]. The text renderer uses FreeType[1] library for TrueType font loading and displays text as a series of textures. One important thing to note, again, is the reversal of the y axis in the shader due to the projection matrix. This has also been taken into account when using the font glyph bearings on the y axis (e.g. vertical offset of the letters 'p' and 'g').

8.3 SkewT/LogP Simulator

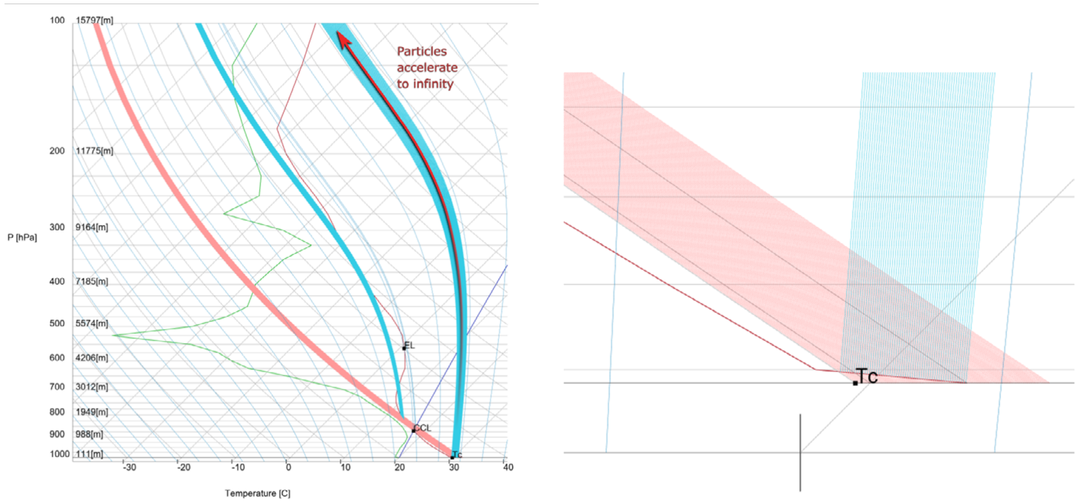
Because the creation of the diagram is a particularly hard problem to parallelize, we create the diagram on the CPU and upload necessary data to the GPU. One caveat here is that we do not assume constant number of curve vertices (or edges) and therefore we need to upload additional information along with the curve data. This is particularly useful in cases where we upload arrays of curves as it is with the dry and moist adiabat profiles. We do not assume constant number of curve vertices since the ground altitude varies greatly among soundings. Furthermore, we do not plot the moist adiabats from ground level considering their generation starts at either CCL or LCL based on parameter selection. This saves us memory on the GPU.

The simulation algorithm isn't particularly too far from the pseudocode shown in Algorithm 1. Only additions are bound checking and artificial damping. Let us look at both and what is the reason behind their usage.

First important addition to the algorithm is out of bound checking. This proved particularly useful for diagrams where unwanted cases occur that lead to particles accelerating to infinity. As an example, see Figure 8.2. There, particles are lifted almost instantly moist-adiabatically. Since the difference between moist adiabats and the ambient temperature curve is always positive in this particular case, the particles would accelerate to infinity very rapidly, producing computation errors in other stages of the application loop (e.g. when sorting using Thrust library). For these reasons we check whether each particle is out of bounds and if its computed direction of acceleration points further out of bounds. In that case we do not update its position and velocity. In instances where the particle is out of bounds but should accelerate towards the simulation area, which happens, for example, when user creates a terrain that has origin below ground pressure in the sounding data, the simulation uses the last valid values to compute the acceleration. This is achieved by clamping the isobar intersection calculation to last valid curve values as shown in Figure 8.3.

---

[1] available at: `https://www.freetype.org/`



(a) Diagram where particles would accelerate to infinity indefinitely.

(b) Closeup of the problematic section.

Figure 8.2: Example of a problematic case where the sounding data produce a diagram that would accelerate particles indefinitely without our additional bound checking.

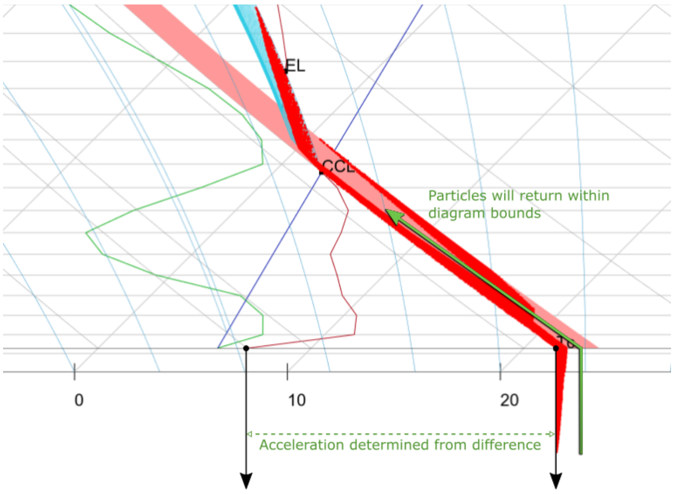

EL
CCL
Particles will return within diagram bounds
Tc
0
10
20
Acceleration determined from difference

Figure 8.3: Visualization of clamping of the isobar intersections to last valid values. Note that the red points represent individual particles in our system.

Unfortunately, there is another issue with the simulation proposed by Duarte for which we did not find any sound explanation. The simulated particles oscillate around the equilibrium level with very large altitude reach. In layman's terms, the particles "overshoot" the equilibrium and stop way above it. After that, they accumulate negative velocity on the y axis, causing them, once again, to "overshoot" the equilibrium. This is particularly visible when choosing $\Delta t < 10$. In our experiments, we observe that the particles actually do reach equilibrium after some time. The smaller Δt is used, the slower the equilibrium is reached. We presume that this is inevitable due to the very nature of the algorithm proposed by Duarte. There is no negative or damping force in either case (when particle is below or above equilibrium) and the particle velocity $\vec{v_i}$ only increases till the equilibrium is reached because only the difference between dry or moist adiabat and the ambient temperature curve is used for acceleration computation (see line 12 in Algorithm 1). To solve this issue, we propose a simple damping multiplier that decreases the particle velocity $\vec{v_i}$ by multiplying it by value $v_{damping} < 1$, $v_{damping} \to 1$.

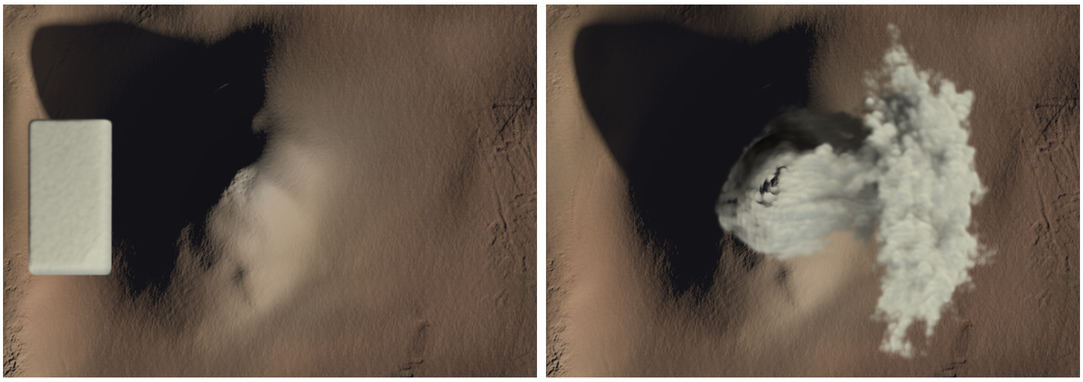
Another problem of Duarte's method is the nature of the SkewT/LogP diagram. The sounding data is observed for a single point in the atmosphere which is heavily dependent on the initial ground altitude/pressure. This means that we would get different results when releasing a radiosonde from top of a mountain and from a valley. Since most weather soundings capture as much information as possible, radiosondes are mostly released from low altitudes to capture larger part of the atmosphere. Thus, the actual simulation for particles that do not start at ground level as described by the sounding data is not physically accurate. This can in some extreme cases result in initial negative acceleration for particles on top of mountain peaks which would move them (moist-adiabatically) downwards through the mountain's boundary. One option to solve this issue is to have areas which use different diagrams based on their altitude. This would however require a lot of memory and for this reason it was not considered in the final implementation.

8.4 Lattice Boltzmann Method

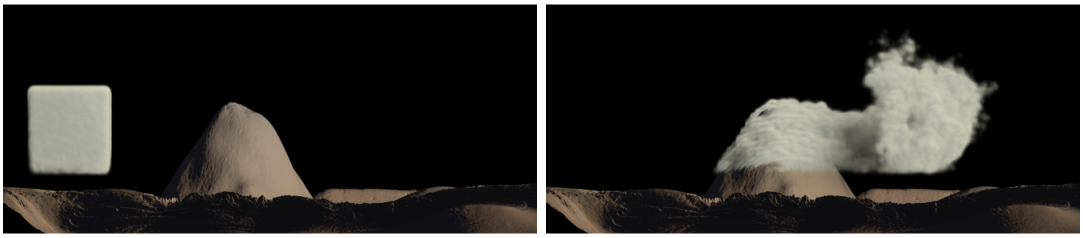
During the creation of this thesis, the Lattice Boltzmann Method (LBM) was implemented both in 2D and 3D on CPU and GPU both. However, the final framework only supports 3D GPU implementation. Once again, because LBM is an easily parallelizable problem, the actual code bears no surprises and does not pose any issues that weren't covered in the theory.

For the simulation, the main idea is to use two lattices where each holds one configuration of the simulation space. The lattices are swapped after each simulation step and the lattice with the configuration from previous step is used for computing a new configuration in the next step. This is similar to using two framebuffers when rendering a window with OpenGL. It is possible to implement indexation that supports in-place update of the lattice as proposed by Latt [Lat07] which could save a large amount of memory for fine lattices and will be taken into consideration in future updates, however, it is now out of scope of this work.

The lattice is stored in memory as a one dimensional array of nodes where each node holds 19 float values in the case of D3Q19 configuration. Furthermore, an array of velocities is also necessary for simulation which results in additional 3 floats per node. As an example, a lattice with resolution 100^3 would require $100^3 \cdot (19+3) \cdot 4$ Bytes which translates to 88 Megabytes of memory.



(a) Top view.



(b) Side view.

Figure 8.4: Box displaced by prebaked LBM simulation displayed with an orthographic projection. Note that we hide particles that are below CCL in the shown images.

Implementation of the individual simulation steps is quite straightforward and can be found in the source file `LBM3D_1D_indices.cu`. Each simulation step that was described in Chapter 5 uses a single kernel for its execution. All kernels are simply called in succession as they appear in Figure 5.6. Note that before particle movement step is done, we need to map the VBOs containing particle positions and profile indices using `cudaGraphicsMapResources` call. After the kernel has been executed, the resource must be freed by calling `cudaGraphicsUnmapResources`. CUDA and OpenGL interoperability is demonstrated on a particle sorting algorithm in Listing 8.6.

Besides the basic steps, we have also implemented the subgrid model approach which ensures that the simulation is stable by recomputing the value of τ in each lattice node. Furthermore, the subgrid model is not dependent on the lattice resolution and can therefore be enabled or disabled at runtime. The model and the theory behind it are further examined in Appendix C. Simulation results using the subgrid model are presented in Figure 8.4.

Since LBM simulation can become unstable even while the subgrid model is enabled, we check that particles have valid values before they are rendered to prevent crashes or other issues. This however requires a kernel that checks all particle positions and is therefore computationally costly.

For the collision step, we make use of shared memory cache that loads the whole block of lattice nodes. This is most effective when the size of blocks is rather small since we want to run a good amount of blocks on a single streaming multiprocessor (SM). This means that we use 32x2 threads in 3D. This configuration of blocks was also used for time measurements.

8.4.1 Standalone Application

In the standalone LBM application, multiple other features were available such as particle color coding based on their velocity as shown in Figure 8.5, particle velocity vector and lattice node velocity vector glyph visualizations, and many more. One interesting detail worth mentioning was the option to switch between CPU and GPU implementations and vice versa at runtime by enabling `LBM_EXPERIMENTAL` and recompiling. Using this switch, users could get a sense of how much faster the GPU implementation is.

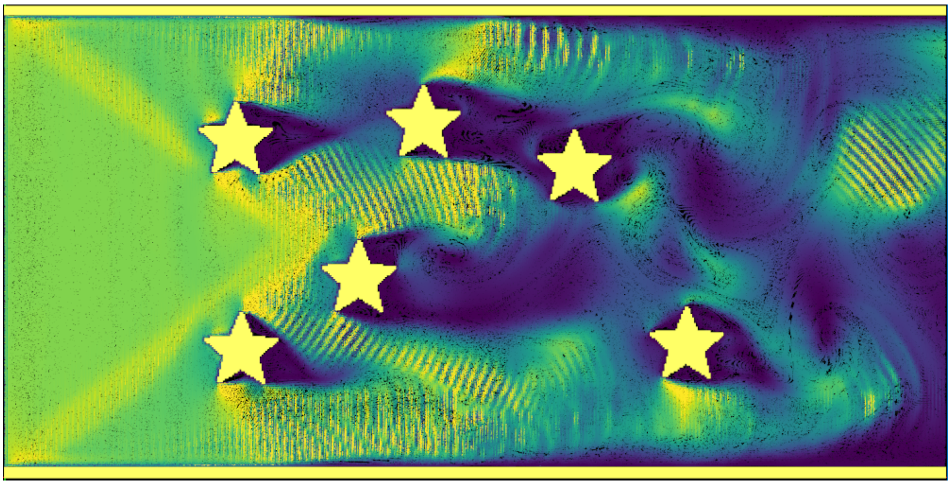


Figure 8.5: Visualization of particle velocities using viridis color map in 2D.

8.4.2 Inlets

In our implementation, we give users a free hand at selecting which walls (faces) of the simulation area bounding box are treated as inlets. The implementation of this system is very simple. The kernel that updates inlets (`updateInletsKernel`) simply checks whether the node of each of the run threads belongs to the given wall. If so, the macroscopic velocity is set to its user defined value and regular collision operator is applied.

8.4.3 Boundaries

Boundaries play a large role in the simulation. Within our application, we offer three possible modes of how particles that cross the simulation area boundaries are treated.

First option is that the particles are simply respawned randomly with uniform distribution in the active inlet wall. Second option respawns particles on the opposite side of the simulation area as if it were a single repeating tile/block. As an example, imagine as if the area was a texture with `GL_TEXTURE_WRAP_S/T` set to `GL_REPEAT`. This is achieved by using a modulo operator. One important detail is that we need to use modulo operating on floats as follows

```
pos.x = fmodf(pos.x + d_latticeWidth - 1, d_latticeWidth - 1)
```

for the particles not to be snapping to lattice nodes on respawn. The third mode is a combination of the previous two where we respawn particles randomly if they leave the boundary on the y axis while cycling them as in the second option on the x and z axes.

8.4.4 Streamline Particle System

Streamlines are an important visualization tool for flow fields that are particularly useful for observing properties of the fluid. Each streamline shows a path taken by a tracked particle starting at specified point. Streamline particle system was implemented for debugging and visualizing our LBM implementation. For this we use a special particle system and LBM kernels that continually create the streamline by activating or reusing predefined sets of vertices. In other words, each streamline has a maximal number of vertices which are all hidden at the start. During the simulation we iterate over this set of vertices and update their position to current position of the particle.

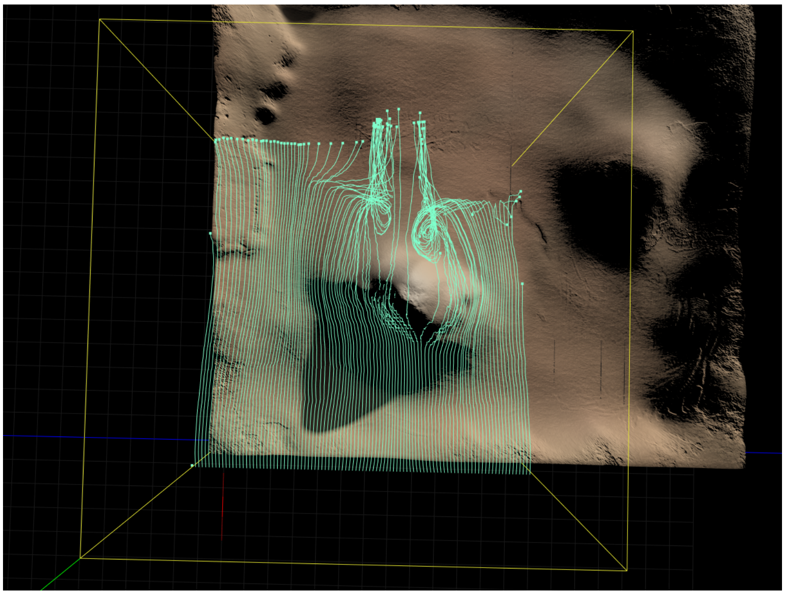


Figure 8.6: Streamlines generated around a peak with $\tau = 0.6$ and the subgrid model enabled demonstrating eddy currents in the fluid flow.

8.5 Cloud Rendering

The cloud rendering algorithm is straightforward and as mentioned before it closely resembles a shadow mapping algorithm with few minor changes. Since the volumetric cloud rendering was a later addition to our system, it is not managed by the `ParticleSystem` class which itself provides a simpler rendering option that draws point sprites with basic blending. The volumetric rendering is handled by a `ParticleRenderer` class.

Before the clouds can be rendered, the half-angle vector must be first calculated and the particles must be sorted. The half-angle vector computation is shown in Algorithm 4.

Algorithm 4: Half-angle vector computation

```
Function recalcVectors(vec3 viewVec, vec3 lightVec)
    if dot(viewVec, lightVec) > 0 then
        halfVec= normalize(viewVec + lightVec);
        invertedRendering= true;
    else
        halfVec= normalize(-viewVec + lightVec);
        invertedRendering= false;
```

8.5.1 Particle Sorting

To achieve real-time speeds, especially with large numbers of particles reaching millions, the sorting must be done quickly. For this, we utilize the GPU and the Thrust[1] library provided by NVIDIA. The sorting step can be decomposed into two substeps. First, we need to compute the projections onto the half-angle vector. This is done by using a custom kernel that goes through all particles (in parallel) and computes the projection as a dot product of their position with the half-angle vector as shown in Listing 8.5. These projection distances are saved to an auxiliary array. In the second step, we utilize Thrust function `sort_by_key` where we use the distances as keys for sorting the particle indices. We use particle indexing with `glDrawElements` (as opposed to `glDrawArrays`) since each particle has multiple properties and we want to avoid sorting all these arrays individually. Furthermore, sorting using a set of keys that have a primitive type value such as float is very beneficial in that Thrust selects a much faster radix sort implementation instead of merge sort implementation that is used on custom objects. The sorting function is shown in Listing 8.6 where CUDA and OpenGL interoperability is demonstrated.

---

[1] available at: `https://thrust.github.io/`

```
__global__ void computeDistances(vec3 *particleVertices, float *particleDistances,
     vec3 sortVector, int numParticles) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < numParticles) {
        particleDistances[idx] = dot(particleVertices[idx], sortVector);
    }
}
```

Listing 8.5: Particle distances computation kernel that uses a projection onto a given sorting vector (half-angle vector in our case).

```
void sortParticlesByProjection(vec3 sortVector) {

    vec3 *d_mappedParticleVerticesVBO;      // mapped particle positions (VBO)
    unsigned int *d_mappedParticlesEBO;     // mapped indices of vertices (EBO)

    cudaGraphicsMapResources(1, &cudaParticleVerticesVBO, 0);
    cudaGraphicsResourceGetMappedPointer((void **)&d_mappedParticleVerticesVBO,
        nullptr, cudaParticleVerticesVBO);

    cudaGraphicsMapResources(1, &cudaParticlesEBO, 0);
    cudaGraphicsResourceGetMappedPointer((void **)&d_mappedParticlesEBO, nullptr,
        cudaParticlesEBO);

    computeParticleProjectedDistances<<<gridDim, blockDim>>>(
        d_mappedParticleVerticesVBO, d_particleDistances, sortVector,
        numActiveParticles); // Listing 8.5

    thrust::sequence(thrust::device, d_mappedParticlesEBO, d_mappedParticlesEBO +
        numActiveParticles); // sequence [0, 1, 2, ..., numActiveParticles]

    // sort active particles using the auxiliary d_particleDistances array
    thrust::sort_by_key(thrust::device, d_particleDistances, d_particleDistances +
         numActiveParticles, d_mappedParticlesEBO, thrust::less_equal<float>());

    cudaGraphicsUnmapResources(1, &cudaParticleVerticesVBO, 0);
    cudaGraphicsUnmapResources(1, &cudaParticlesEBO, 0);
}
```

Listing 8.6: Particle sorting algorithm showcasing VBO and EBO mapping with a kernel call.

8.5.2 Slice Rendering

The clouds are rendered in two passes for each slice. For this, two render targets (framebuffers) are needed. One for rendering the cloud particles from the light's point of view and one for rendering the particles from the camera's perspective. Let us call the first framebuffer a *light framebuffer* and the second an *image framebuffer*. We iterate through all slices and accumulate particle lightness into the light framebuffer which is then used as a texture sampler for when rendering into the image framebuffer. This process is shown in Algorithm 5. Note that for each slice we have to switch the render target twice which is computationally expensive. Blurring of the light texture in-between rendering individual slices is also possible to give an impression of more scattered light within the medium. This also gives softer shadows if the light texture is used for shadow mapping.

Algorithm 5: Drawing slices iteratively

```
Function drawSlices()
    batchSize ← ceil(numActiveParticles / numSlices);
    clear light framebuffer with vec4(vec3(1.0) - dirLight.color, 0.0);
    clear image framebuffer with vec4(0.0);
    for i = 0 to numSlices - 1 do
        drawSlice(i);
        drawSliceLightView(i);
        if useBlurPass then
            blurLightTexture();
```

The blending of particles also depends on the current pass of the algorithm. When rendering to the image framebuffer, blending varies based on the half-angle vector computation as shown in Algorithm 6. When rendering the particles to the light buffer, (`GL_ONE`, `GL_ONE_MINUS_SRC_COLOR`) blending is utilized as shown in Algorithm 7.

Algorithm 6: Drawing a slice to the image framebuffer

```
Function drawSlice(int i)
    bind image framebuffer;
    set viewport to screen size;
    if invertedRendering then
        blend function ← (GL_ONE_MINUS_DST_ALPHA, GL_ONE);
    else
        blend function ← (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
    drawPointSprites(imagePassShader, i · batchSize, batchSize, true);
```

Algorithm 7: Drawing a slice to the lighting framebuffer

```
Function drawSliceLightView(int $i$)
    bind light framebuffer;
    set viewport to light texture resolution;
    blend function ← (GL_ONE, GL_ONE_MINUS_SRC_COLOR);
    drawPointSprites($lightPassShader$, $i \cdot batchSize$, $batchSize$, false);
```

Initially, we chose to draw particles as simple point sprites using OpenGL's built-in texture coordinates (`gl_PointCoord`) for texturing. This later proved to be problematic since we draw the particles from two points of view with different projection matrices. This means that even if we were to scale the points according to either distance to camera or distance to light, we would get shading that is constantly changing. In other words, we need control over the world size of the quad that is drawn. For this, we use a simple geometry shader in both passes which gives us full control over the particle rendering at a small cost in performance. We show a pseudocode of the remainder of the described rendering process in Appendix E.

8.5.3 Auxiliary Framebuffers

A very important part of our rendering pipeline is the usage of auxiliary framebuffers for rendering the scene. As any reader with some experience with graphics programming would assume, we choose to use auxiliary framebuffers for post-processing, HDR and tone-mapping purposes. This is true to some degree, but we would like to stress that the main reason this approach was selected pertains to cloud rendering. Since we render the clouds separately to a custom framebuffer, we need depth data generated when rendering opaque objects (or more generally, any objects that were rendered before clouds and should occlude them). As shown in the sample implementation by Green, one option is to draw the scene depth data to the framebuffer that is used for rendering final cloud color. This is however a problem when rendering complex scenes as is our case where large terrain and instanced meshes such as trees are present. With this approach, the scene would have to be drawn twice to two depth buffers separately.

To solve this issue, using an auxiliary framebuffer object is a simple and effective solution. We draw the scene to the auxiliary framebuffer, let us call it the main framebuffer of our application. Then, when we render the cloud particles, we attach the depth component texture of our main framebuffer to the cloud image framebuffer. This is safe even if we wanted to use the depth attachment later for other tests since the cloud rendering algorithm does not write to the depth buffer, it only reads from it. There is one drawback however, because we use the same depth attachment for both the main framebuffer and the cloud image framebuffer, both their resolutions must be the same. This is only an issue if we want to render the clouds with a lower resolution than the final image to reduce computational complexity.

Additionally, we want our system to utilize multisample anti-aliasing (MSAA). Second framebuffer that has multisampled color and depth attachments is needed for MSAA to be possible. We render the scene to the multisampled framebuffer. Before rendering the clouds, we blit the multisampled color and depth values to the regular auxiliary framebuffer to use the regular depth texture in cloud rendering where MSAA is unnecessary. From then on we use the regular auxiliary framebuffer till the end of rendering loop at which we transfer its color data to the default window framebuffer. The whole process is illustrated in Figure 8.7.

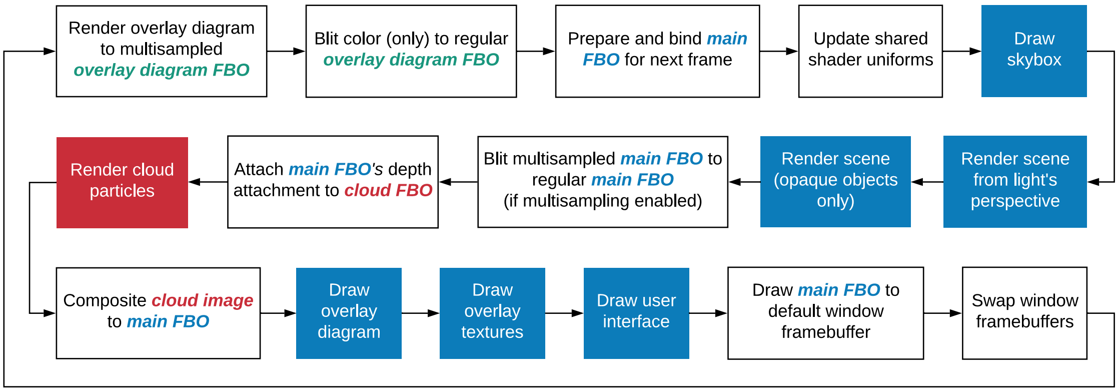

Render overlay diagram to multisampled ***overlay diagram FBO***
Blit color (only) to regular ***overlay diagram FBO***
Prepare and bind ***main FBO*** for next frame
Update shared shader uniforms
Draw skybox
Render scene from light's perspective
Render scene (opaque objects only)
Blit multisampled ***main FBO*** to regular ***main FBO*** (if multisampling enabled)
Attach ***main FBO***'s depth attachment to ***cloud FBO***
Render cloud particles
Composite ***cloud image*** to ***main FBO***
Draw overlay diagram
Draw overlay textures
Draw user interface
Draw ***main FBO*** to default window framebuffer
Swap window framebuffers

Figure 8.7: Rendering loop including the usage of auxiliary framebuffers.

8.5.4 Phase Function Integration

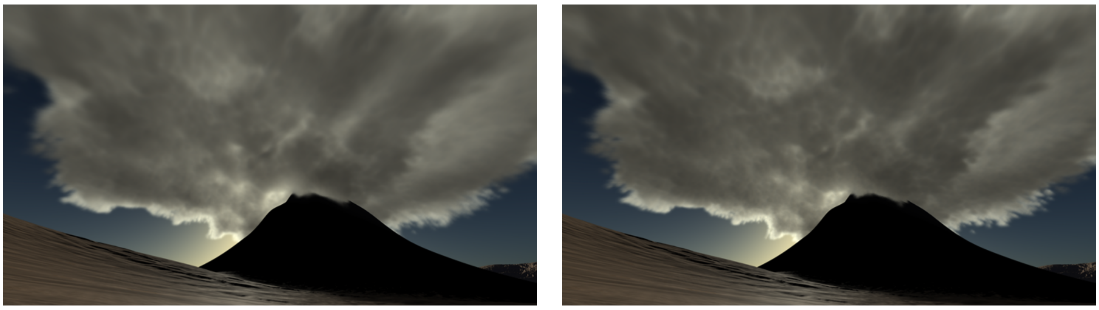
We have implemented all the phase functions presented in Section 6.6 into our framework. Users can therefore alter between them at runtime to observe their properties. Since we do not use a physically-based volume rendering approach, we incorporated the phase functions using a simple trick. During the rendering of each particle batch, we increase the intensity of the fragment as follows

```
fragColor.xyz *= (vec3(1.0)+ shadow * phaseFunctionVal)
```

Multiplying the output of the phase function with the shadow intensity makes sure that the increased intensity is only visible in less dense areas. Since we want to increase the intensity only, we need to add unit vector to the multiplication term.

8.5.5 Performance Note

One very important thing to note is the setting of individual parameters and the number of particles used. One may obtain similar results with much lower amount of particles by increasing their opacity and size, which usually yields much faster rendering. This depends on scene and complexity of the shape formed by the particles. In Figure 8.8 we show a comparison between two cloudscapes generated with different sets of parameters producing visually similar results.



(a) Clouds composed of 500k particles with opacity multiplier set to 0.1.

(b) Clouds composed of 200k particles with opacity multiplier set to 0.3 and with more intense blur.

Figure 8.8: Comparison of clouds with different visualization parameter settings and particle counts.

8.5.6 Stylization

As part of experimentation we have implemented the possibility of using simple atlas textures to further randomize the visual appearance of rendered clouds. Furthermore, the cloud rendering algorithm is quite suitable for stylized visualization of clouds as shown in Figure 8.9

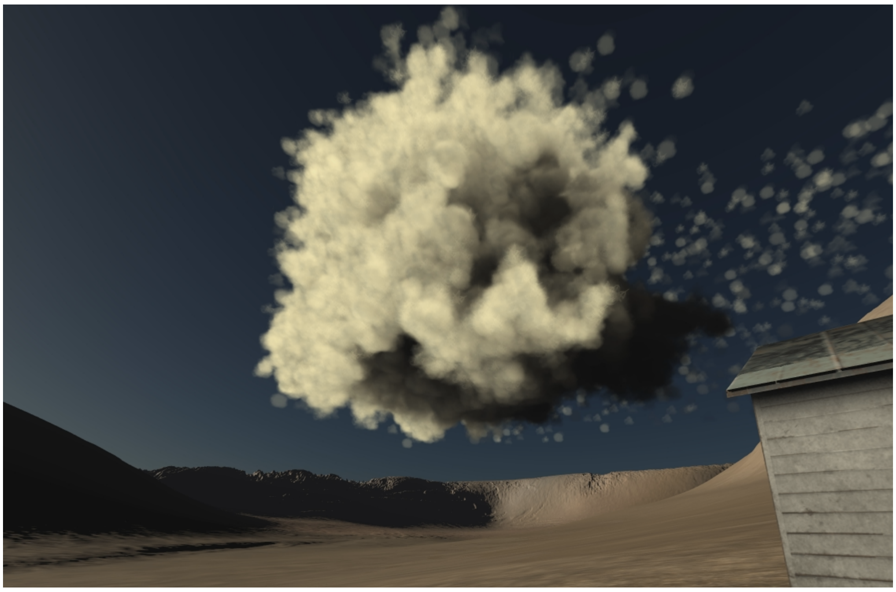


Figure 8.9: Stylized appearance of clouds using a texture atlas to select different sprite textures randomly.

8.6 Sky Rendering

Large part of the final impression from any generated image is its backdrop/background. In our case the most important background is the sky atmosphere itself. Since we want to present our clouds in dynamic environments, HDRI sky images are not very practical since they are static and usually contain clouds themselves (which is not necessarily a problem, especially if the clouds represent the high clouds from upper parts of the atmosphere). That is why we chose to use a dynamic sky model, particularly the one proposed by Hošek and Wilkie [HW12]. An implementation of the model is public on their webpage[1], however, we have based our code on an implementation by Ben Anderson that uses the Rust language which is available on his GitHub page[2] [And17].

The Hošek-Wilkie model is an improvement of the Preetham model [PSS99] based on the same formula by Perez et al.[PSM93]. The Perez formula for describing the luminance of clear skies is defined as

$$\mathbb{F}_{Perez}(\theta,\gamma) = (1 + Ae^{B/\cos\theta})(1 + Ce^{D\gamma} + E\cos^2\gamma) \tag{8.18}$$

where γ is the angle formed by the view direction and a vector pointing towards the sun, and θ is the angle between zenith and view direction as shown in Figure 8.10 [HW12].

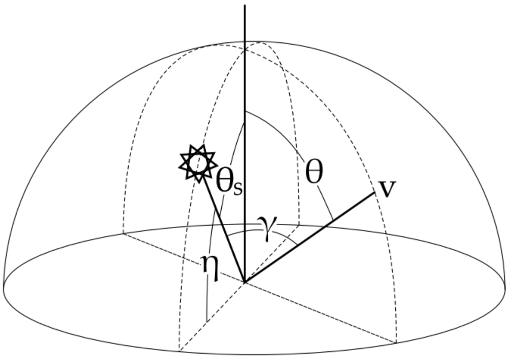


Figure 8.10: Angles used in the computation. η is the elevation while θ and γ are used to compute the fragment color and are plugged into the updated Perez formula [HW12].

In Hošek-Wilkie's model, a path tracer was used to produce realistic sky renders that are then fitted using Levenberg-Marquardt non-linear least-square method in MATLAB. In their source code, they provide data for their original spectral bands, CIE XYZ model and RGB model. The updated Perez formula is defined as

$$\mathbb{F}(\theta,\gamma) = (1 + Ae^{\frac{B}{\cos\theta + 0.01}}) \cdot (C + De^{E\gamma} + F\cos^2\gamma + G \cdot \chi(H,\gamma) + I \cdot \cos^{\frac{1}{2}}\theta) \tag{8.19}$$

$$\chi(g,\alpha) = \frac{1 + \cos^2\alpha}{(1 + g^2 - 2g\cdot\cos\alpha)^{\frac{3}{2}}} \tag{8.20}$$

---

[1] available at: `https://cgg.mff.cuni.cz/projects/SkylightModelling/`

[2] available at: `https://github.com/benanders/Hosek-Wilkie`

We implement the formula on both the CPU and GPU. On CPU, the formula equations are necessary for parameter recalculation when any angles (θ, γ), or any parameters such as turbidity of the sky or ground albedo, change. The GPU implementation is in a form of a fragment shader (`sky_hosek.frag`) that renders the sky using a simple cube. Example of the sky model in use is shown in Figure 8.11.

As was mentioned, even though HDRI images aren't ideal for our purposes, they are a stepping stone to the dynamic sky model. For this reason, their implementation is also available in our framework.

To improve visual fidelity of our engine, we wish to update sun color based on the appearance of the atmosphere. For this we utilize a simple approach of sampling the atmosphere on the CPU and updating the sun light color. Since the atmosphere generates very saturated colors when the sun elevation is low, we interpolate the final color as a mix of white color and the sky sample. The interpolation coefficient can be modified by the user at any time.

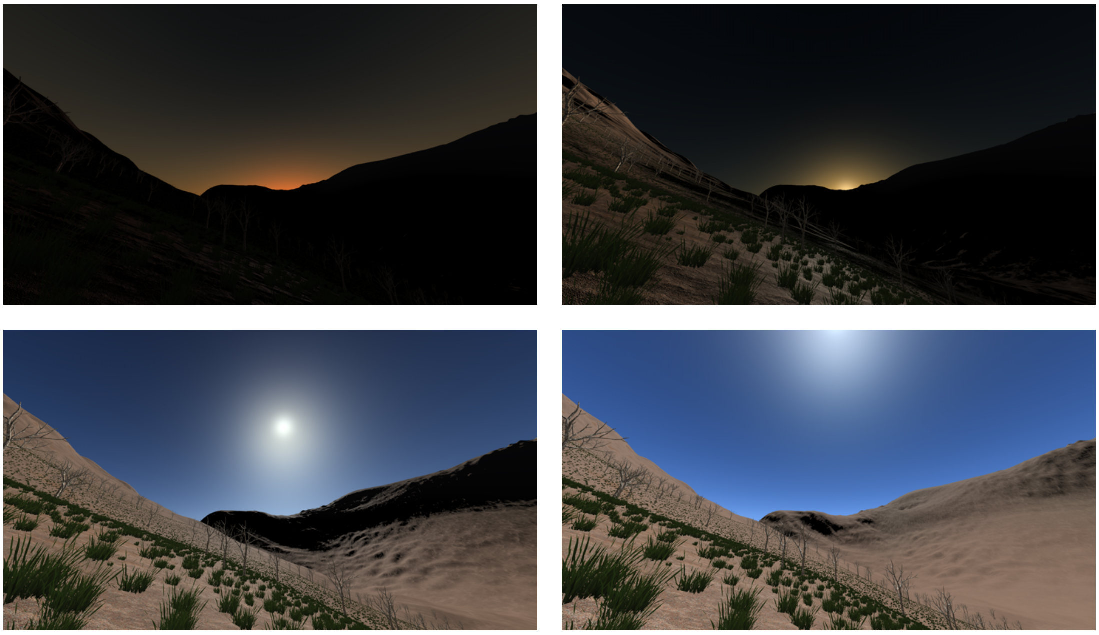


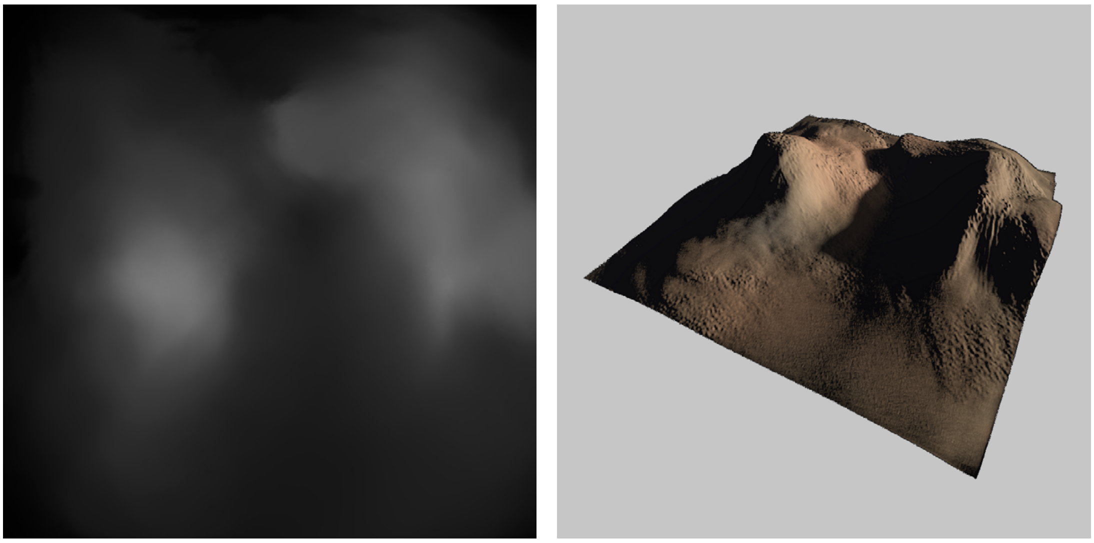
Figure 8.11: Sunrise generated with the Hošek-Wilkie model with turbidity set to 4 and albedo to 0.5.

8.7 Terrain

Terrain plays a major part in our application since it determines where the particles can be generated and which nodes of the lattice are obstacle nodes. All this information is stored in the heightmap from which the terrain is created at the initialization stage of the application. Since it is not the main focus of this thesis, the terrain is created naively on the CPU.

8.7.1 Heightmap

As mentioned above, the terrain is defined by its heightmap. In our case it is generally a grayscale texture where each texel describes height of the terrain within a given range. This range can be selected based on multiple factors, but it generally depends on the terrain scale. For loading the heightmap, we use a single header library `stb_image.h` [1] which gives us the ability to load 16-bit per channel textures. This is particularly useful because we want to create a smooth terrain and 8-bit information does not suffice considering it only allows 256 height levels. There is an option of using all RGBA channels instead of a grayscale image with additional logic that would assign a scale factor to each channel. For example values in the red channel would be in units of hundreds meters, G in tens of meters and so on. Problem of this approach is mainly that the creation of a custom heightmap is not intuitive and the user cannot determine the terrain profile just from looking at the texture. An example of the grayscale image and its corresponding terrain is shown in Figure 8.12.



(a) Heightmap used for terrain generation.

(b) Terrain created from the heightmap.

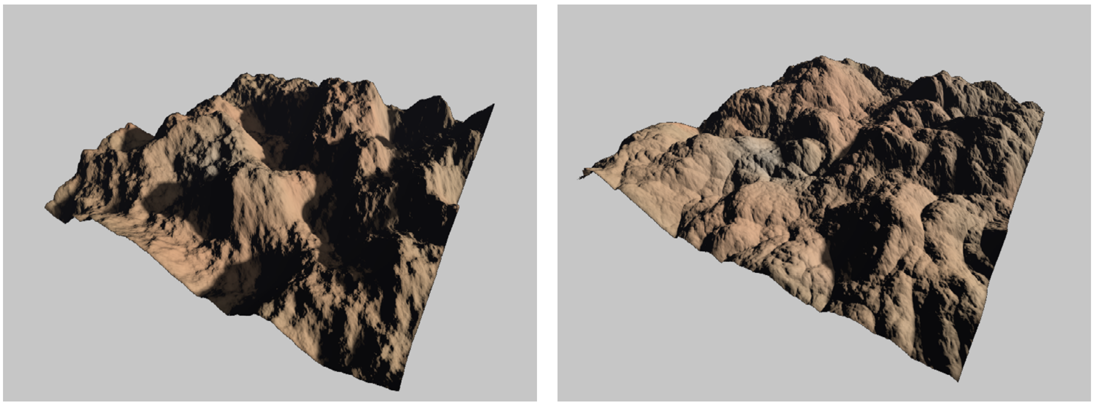
Figure 8.12: Grayscale heightmap used for terrain generation and the resulting terrain.

---

[1] available at: `https://github.com/nothings/stb`

8.7.2 Perlin Noise

Additionaly, a perlin noise generator was implemented based on Perlin's Java reference source code for his improved perlin noise [Per02b, Per02a]. We have extended his reference code with octaves as shown in Listing 8.7 and an option to generate turbulent noise which takes the absolute value of each sample. The sampler can be used statically or as an instance where settings for frequency, number of octaves, sampling mode and more are remembered for each individual sampler. Perlin noise is also utilized in particle emitters where it is fed into CDF samplers as will be described in Section 8.8.



(a) General perlin noise used.

(b) Turbulent perlin noise used.

Figure 8.13: Example of terrains generated by perlin noise with 10 octaves.

```
float perlinNoiseSampleOctaves(float x, float y, float z, float startFrequency,
    int numOctaves, float persistence, int samplingMode) {
    if (numOctaves <= 1) {
        return perlinNoiseSample(x, y, z, startFrequency, samplingMode);
    }
    float frequency = startFrequency;
    float amplitude = 1.0f;
    float maxTotalValue = 0.0f;
    float totalValue = 0.0f;
    for (int i = 0; i < numOctaves; i++) {
        totalValue += perlinNoiseSample(x, y, z, frequency, samplingMode) *
            amplitude;
        maxTotalValue += amplitude;
        amplitude *= persistence;
        frequency *= 2.0f;
    }
    return (totalValue / maxTotalValue);
}
```

Listing 8.7: Generating perlin noise with multiple octaves.

8.7.3 Creation

We create the terrain by iterating through the heightmap texture and creating quads composed of two triangles for each 4 adjacent texels. To create a smooth appearance we need to generate interpolated normals for the terrain mesh. One option would be to iterate over all terrain vertices and interpolate normals of incident triangles. This is not necessary since all the information required to create normals for smooth shading is already stored in the heightmap.

For computing the normals from heightmap, the trick lies in the fact that we can compute the tangents in x and z directions (assuming that y axis points upward as is in OpenGL). The tangent vector on the x axis of the terrain surface can be computed as

$$\vec{t_x} = \left(\frac{\partial x}{\partial x}, \frac{\partial h(x,z)}{\partial x}, \frac{\partial z}{\partial x}\right) = \left(1, \frac{\partial h(x,z)}{\partial x}, 0\right) \tag{8.21}$$

where $y = h(x, z)$ denotes the height function. Similarly, the tangent vector on the z axis is

$$\vec{t_z} = \left(\frac{\partial x}{\partial z}, \frac{\partial h(x,z)}{\partial z}, \frac{\partial z}{\partial z}\right) = \left(0, \frac{\partial h(x,z)}{\partial x}, 1\right) \tag{8.22}$$

[Daw18]

For estimating the partial derivatives, central difference formula is used [MF$^+$04]. Assume that $f \in \mathrm{R}^3[a, b]$ and that $x - h,\ x,\ x + h\ \in\ [a, b]$. Then

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{8.23}$$

This means that we can estimate

$$\frac{\partial h}{\partial x} \approx \frac{R - L}{2} \quad \text{and} \quad \frac{\partial h}{\partial z} \approx \frac{T - B}{2} \tag{8.24}$$

where R, L, T, B denote the right, left, top, and bottom adjacent heightmap texels.

To obtain the final normal vector, we compute a cross product taking the right-hand rule into account as

$$\vec{n} = \vec{t_z} \times \vec{t_x} = \left(-\frac{R - L}{2}, 1, -\frac{T - B}{2}\right) = (L - R, 2, B - T) \tag{8.25}$$

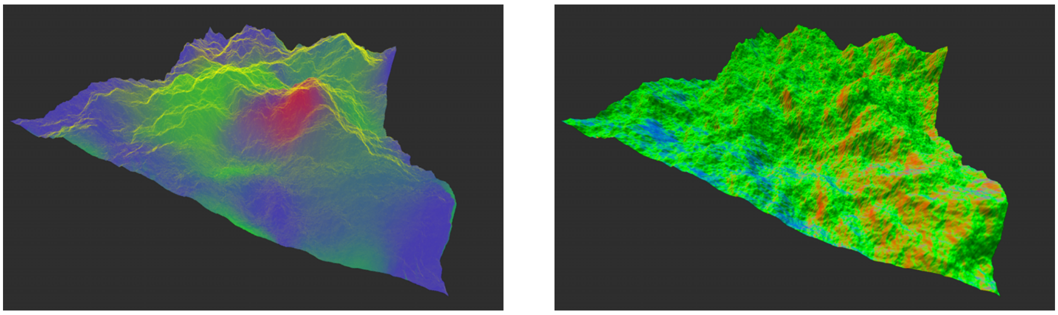
Implementation of this can be seen in Listing 8.8. Note that we divide the tangents with the texel size in world units and normalize the resulting vector.

```
vec3 computeNormal(int x, int z) {
    int xLeft   = max(x - 1, 0);
    int xRight  = min(x + 1, width - 1);
    int zBottom = max(z - 1, 0);
    int zTop    = min(z + 1, depth - 1);

    float heightLeft   = heightData[xLeft][z];
    float heightRight  = heightData[xRight][z];
    float heightBottom = heightData[x][zBottom];
    float heightTop    = heightData[x][zTop];

    vec3 normal;
    normal.x = (heightLeft - heightRight) / texelWorldSize;
    normal.y = 2.0f;
    normal.z = (heightTop - heightBottom) / texelWorldSize;

    return normalize(normal);
}
```

Listing 8.8: Calculating normal vector from height data.

8.7.4 Texturing

Our framework supports multiple textures/materials on the terrain by using multiple texture units in a special purpose shader set just for terrain (`terrain_pbr.vert` and `terrain_pbr.frag`). To determine how the materials are mixed together, an RGBA image called the material map is used. Each of its channels determines how much each material contributes to the final color. The same goes for other properties such as normal maps, metalness, roughness, and ambient occlusion (AO). Our framework also supports visualization of any textures on the terrain. In Figure 8.14 we demonstrate this feature by rendering the terrain with its material map (a) or with its normal map (b).



(a) Material map with normals shown using a geometry shader.

(b) Visualized normal maps.

Figure 8.14: Visualizations of textures on our terrain.

8.8 Emitters

Another important part of the cloud generation are emitters that are placed on the ground. Without LBM, shapes of the emitters are the main contributor to the final cloud shape. We have taken a very user-oriented approach in regards to emitter usage. There are two main categories of emitters present in our framework: emitters that span the whole terrain and so-called positional emitters that can be placed anywhere in the world as shown in Figure 8.15.

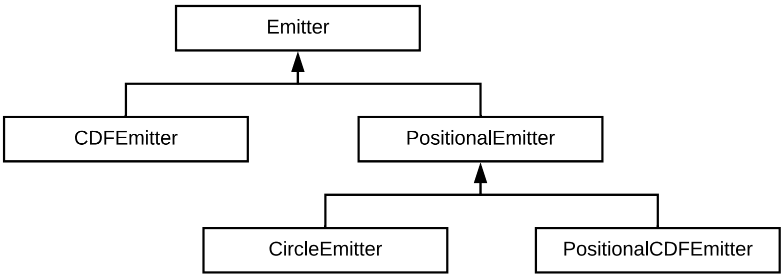


Figure 8.15: Class diagram of implemented emitters.

Air parcel particles are heated on the ground based on multitude of factors in real life. To that end, simple geometric shapes such as circles and squares aren't enough to capture fascinating shapes that a cloud can form when viewed from above. Optimally, emitters could have any shape imaginable. For this reason we have chosen to implement cumulative distribution function (CDF) based sampling giving us the option to sample from a given probability texture. With this approach, any grayscale texture can be used as a pattern delineating an emitter area on the terrain for drawing clouds. Users can therefore create custom shapes that they may wish to reproduce in their favorite digital art software. A good example of this can be seen in Figure 8.16 where a black and white logo was taken to create a desired cloud shape.

8.8.1 Cumulative Distribution Function Sampling

The cumulative distribution function (CDF) $F_X : \mathrm{R} \rightarrow [0,1]$ is a basic concept from theory of probability and statistics. It is defined as

$$F_X(t) = P(X \leq t) \tag{8.26}$$

where $P(X \leq t)$ is the probability that the random variable X has a value less than or equal to t [Nav07]. We sample the 2D image by using the inversion method. This means that we generate uniform random variables and map them to the random variables from the desired distribution, in this case a 2D probability image.

The CDF is computed on the CPU when we load the texture. If we want to modify the texture contents at runtime, we upload the image data to GPU and update the sums array (that represents the CDF) using Thrust library's inclusive scan as described by

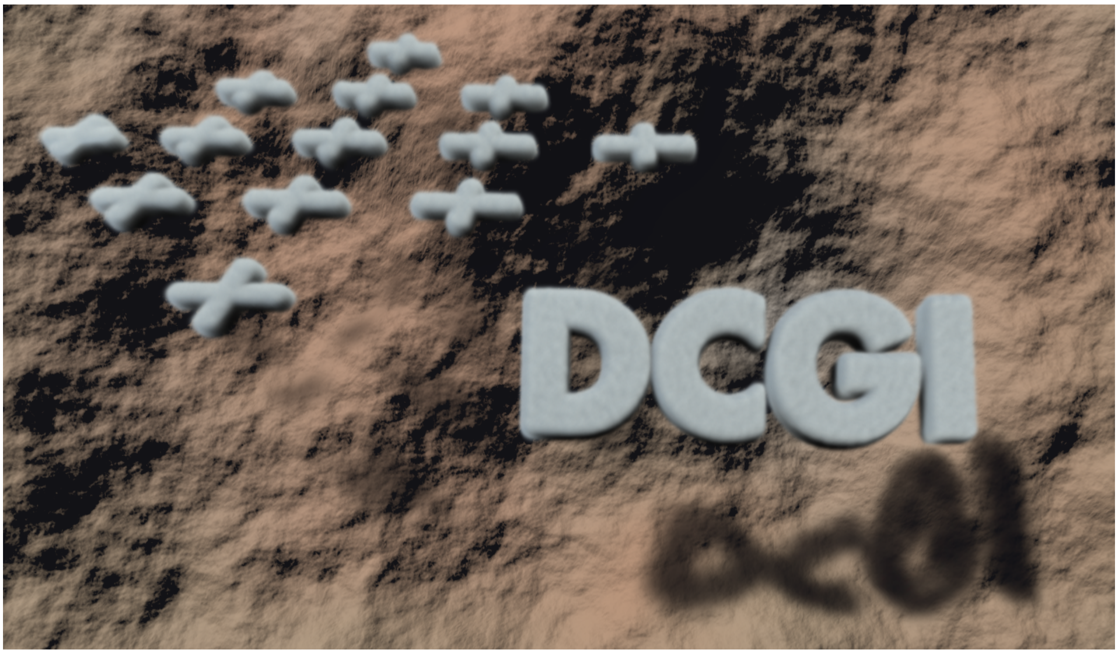


Figure 8.16: Cloud formation shaped as the logo of the Department of Computer Graphics and Interaction created with emitter that uses CDF sampling. Also note the cloud shadow generated by our method.

Barák et al. [BBH13]. Our approach differs from conventional approaches in the fact that we do not normalize the function to range [0, 1] since we load the texture as a set of discrete values of type unsigned short int. This means that when we generate a uniform random sample for an array, we generate it in range [1, *totalSum*] where *totalSum* is the last value in the array after the inclusive scan was computed. Furthermore, as opposed to Barák et al., we flatten the probability image into a 1D array. This is done particularly to reduce the number of kernel calls and generally simplify the procedure. The actual sampling function is shown in Algorithm 8.

Since we are able to generate any given shape on the terrain using the inverse CDF mapping, we can also generate particles using any desired noise function. As an example, we provide an option to use perlin noise in tandem with CDF sampler to either multiply an existing texture or use the perlin noise output directly for sampling as shown in Figure 8.17.

8.8.2 Brush Mode

Placing and moving emitters using the user interface is a cumbersome process and even though our framework offers CDF sampling, we felt that a more user friendly approach was still needed. For this reason, a brush mode for emitters was implemented. The brush mode builds on the concept of positional emitters by using them as brushes. `EmitterBrushMode` class manages brushes and determines which brush is currently active.

Algorithm 8: CDF sampling procedure

```
**Data:** $sums$ array containing inclusive prefix sums of the probability texture
**Result:** $row$ and $col$ indices of the sample
$val \leftarrow$ random value $\in [1, totalSum]$
$L \leftarrow 0$
$R \leftarrow n - 1$
while $L \leq R$ do
    $idx \leftarrow \lfloor \frac{L+R}{2} \rfloor$
    if $val \leq sums[idx]$ then
        $R \leftarrow idx - 1$
    else
        $L \leftarrow idx + 1$
$row \leftarrow L \; / \; width$
$col \leftarrow L \; \% \; width$
```

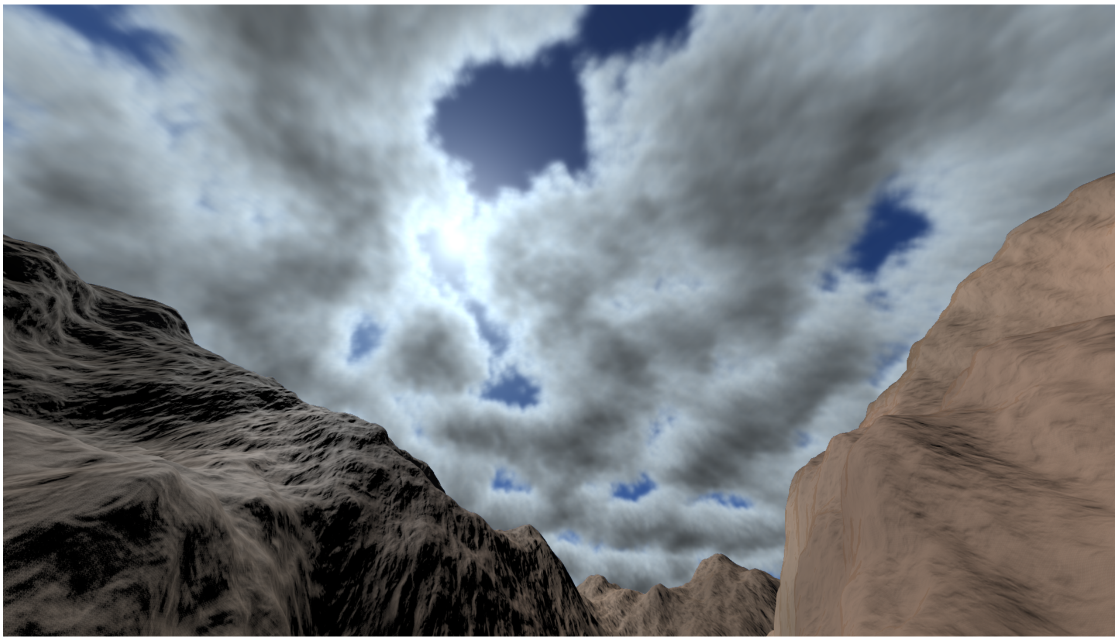


Figure 8.17: Clouds generated with perlin noise that was sampled using CDF with no wind applied.

If the brush mode is enabled, `EmitterBrushMode` is an aggregate class of `ParticleSystem` that handles mouse input events by placing the active brush where the mouse cursor intersects the terrain. For pixel perfect selection of the terrain point intersected by the mouse cursor, we use a `TerrainPicker` class.

The `TerrainPicker` class has its own framebuffer for rendering the terrain with custom fragment shader that renders the world space position of each fragment to the framebuffer's color attachment. For this to be valid, the color attachment must be of type `GL_FLOAT`, otherwise OpenGL would clamp the values to a range [0, 1]. The terrain picker must be updated in each frame by rendering the terrain when the brush mode is active and the current camera was moved. Obtaining the world space position of terrain pixel selected by the mouse cursor is then as simple as calling `glReadPixels(mouse_x , mouse_y, ...)`. Example of a hand drawn cloud using the brush mode is shown in Figure 8.18.

The brush mode works with simple emitters as well as with more advanced positional CDF emitters giving users countless options to draw their cloudscapes. Lastly, the emitter brush mode also supports scaling the brushes, changing their opacity (how quickly they emit particles) or changing profile indices of the emitted particles.

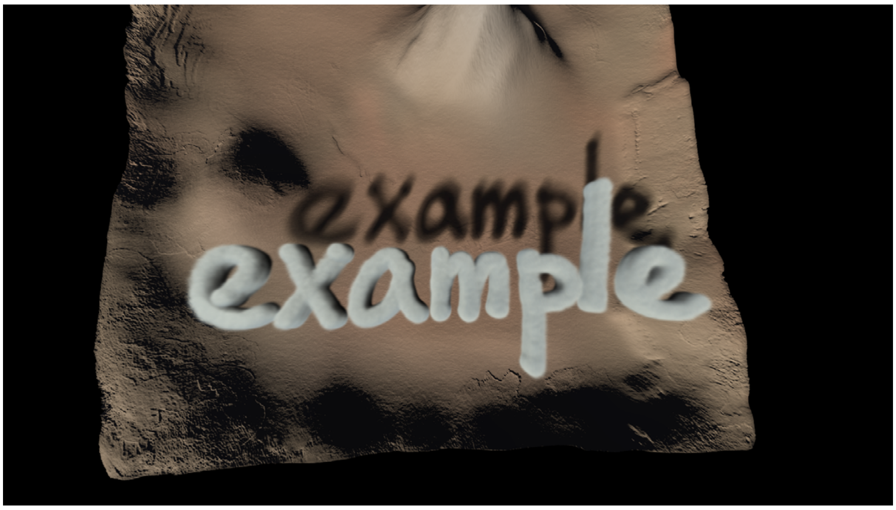


Figure 8.18: Example of a hand drawn cloud shape using a basic circular emitter.

8.8.3 Memory Management

Careful reader may wonder how the emitters upload data to the GPU without degrading the performance. Uploading individual particles would be very costly and not viable if we were to generate ten thousand particles in one frame which is permitted in our system. To solve this issue, the `ParticleSystem` calls all its emitters in its `update` function and accumulates emitted particle data for the current frame. Particles are then uploaded in a batch to the end of the active particles block if there is remaining memory left.

8.9 General

Shading Model. The main building stone of any engine is its shading model. Initially, we have opted to use only a basic Blinn-Phong implementation due to its simplicity. As the application evolved, we've decided that adding simple physically-based rendering (PBR) materials[1] would improve the visual fidelity of our application greatly. Due to this change of heart, the application supports rendering using both approaches.

Screen Size. We would like to note that our framework is designed with screen resize and fullscreen mode in mind. For this, each system that uses auxiliary framebuffers for rendering has to be refreshed at screen resolution change. Furthermore, in case of minimized window where screen width and height are set to zero, the application yields until it is reopened. Correct screen aspect ratio is also managed by refreshing necessary projection matrices.

User Interface. An important part of our framework is its user interface that gives users freedom in customizing individual parts of the simulation process. Nuklear library [Met17] has been selected to create a simple and user-friendly graphical interface. Nuklear is an ANSI C single header library that offers good functionality and customizability. It is an immediate mode GUI which gives us easy control over the individual events generated by users. For more information about the user interface, please see Appendix F.

Application Loop. When all the features described in this chapter are put together, we get a comprehensive framework that must update and render a wide-range of systems. In Figure 8.19 we present a very simplified application loop of the framework. Note that the majority of presented steps can be enabled/disabled. Only exceptions are the initialization stage and opaque object rendering.

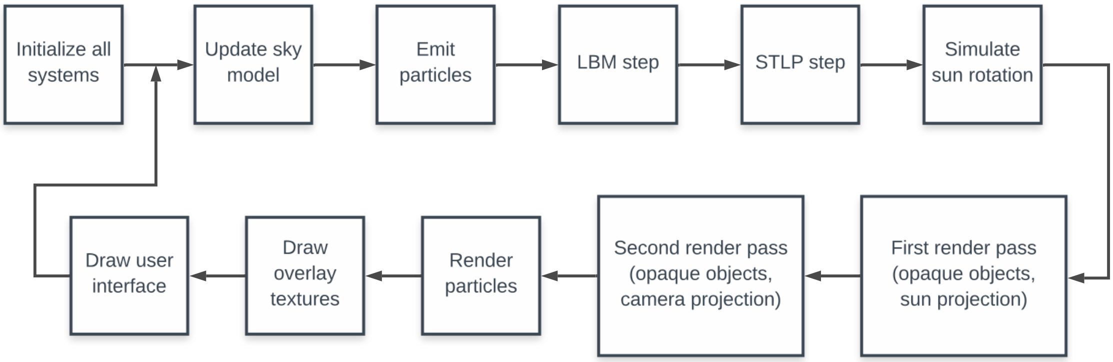

Initialize all systems
Update sky model
Emit particles
LBM step
STLP step
Simulate sun rotation
First render pass (opaque objects, sun projection)
Second render pass (opaque objects, camera projection)
Render particles
Draw overlay textures
Draw user interface

Figure 8.19: Main loop of the application.

---

[1] PBR implementation based on: `https://learnopengl.com/PBR/Lighting`

9 Results

In this chapter we would like to present some of the results obtained when using our framework. One of our main focuses was making all systems interactive and making all the simulation steps optional and fully configurable. Thanks to this, fundamentally different results can be obtained with a little bit of experimentation. In Figure 9.1 fog-like clouds seen from an observer's point of view on top of a mountain are shown. This image was generated with lower opacity particles flowing through the mountain range using both simulation methods. In Figure 9.2 the reader can observe an overcast sky generated using multiple perlin noise emitters and the STLP method. Furthermore, in Figure 9.3 a thick layer of clouds displaced by a mountain peak is shown.

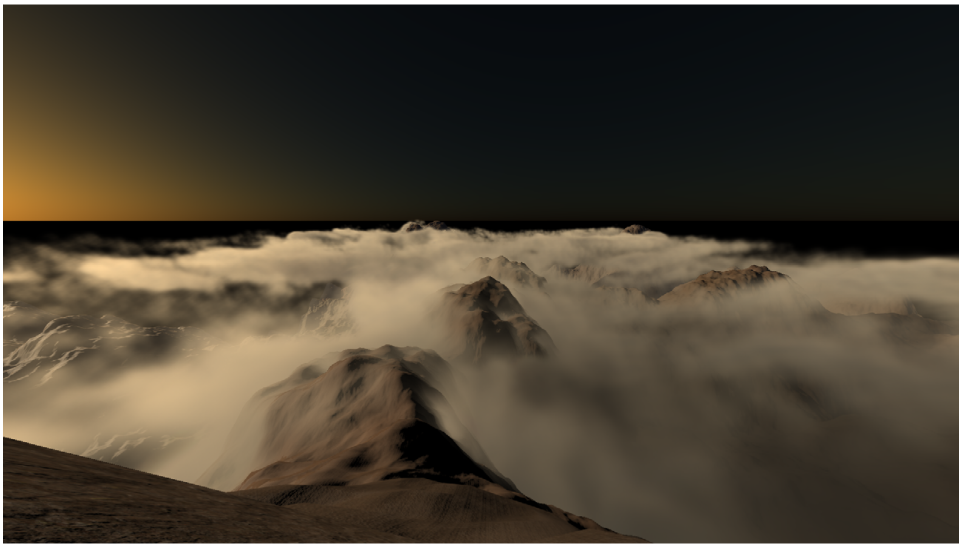


Figure 9.1: Fog-like clouds flowing through a mountain range.

Comparison of our method with a reference photograph can be seen in Figure 9.4 where the clouds are flowing through a mountainous terrain with high peaks. Another comparison showcasing the visual properties of our cloud rendering method is presented in Figure 9.5. In it, a single cloud was rendered with high particle shadowing multiplier generating the very dark appearance as seen in the reference photograph. Third comparison (Figure 9.6) illustrates how our method generates and renders a thick layer of clouds at dawn. In Figure 9.8 we demonstrate how a simple box of particles can be scattered using our simulator to generate an interesting cumulus cloud next to a hill. Lastly, comparison of our overcast stratocumulus clouds with a reference photograph is shown in Figure 9.9.

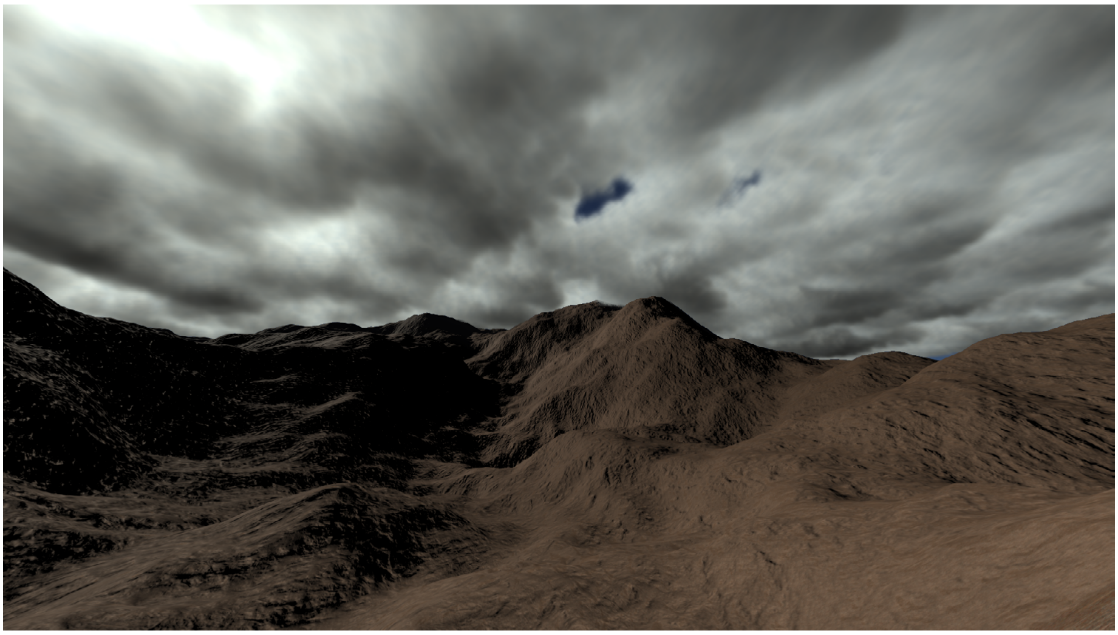


Figure 9.2: Clouds generated with multiple perlin noise emitters showing a cloud overcast with stratocumulus-like appearance.

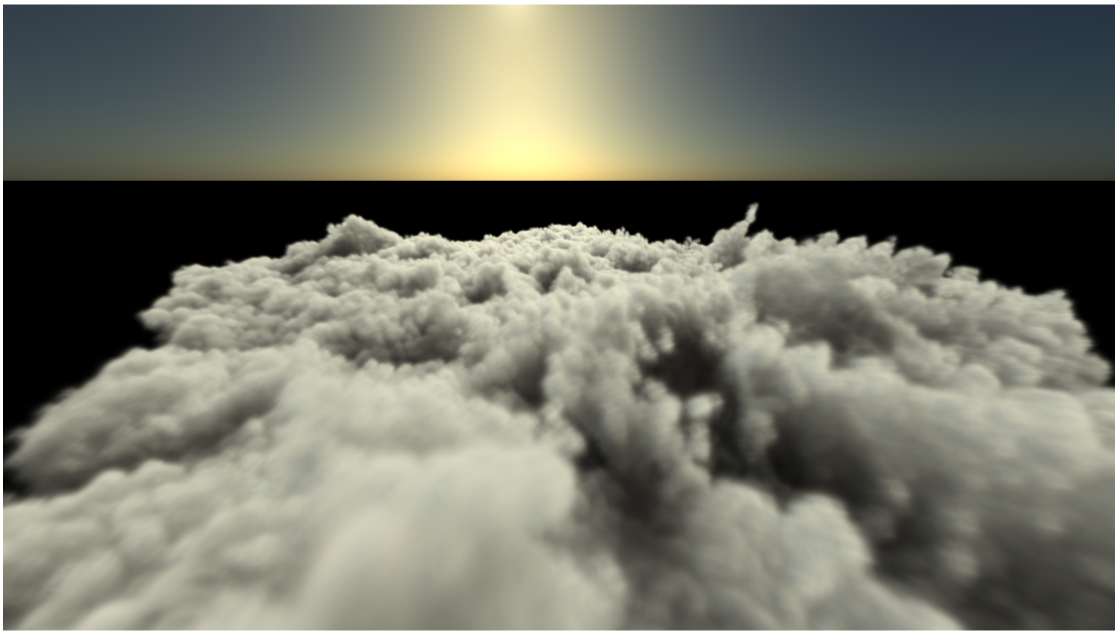
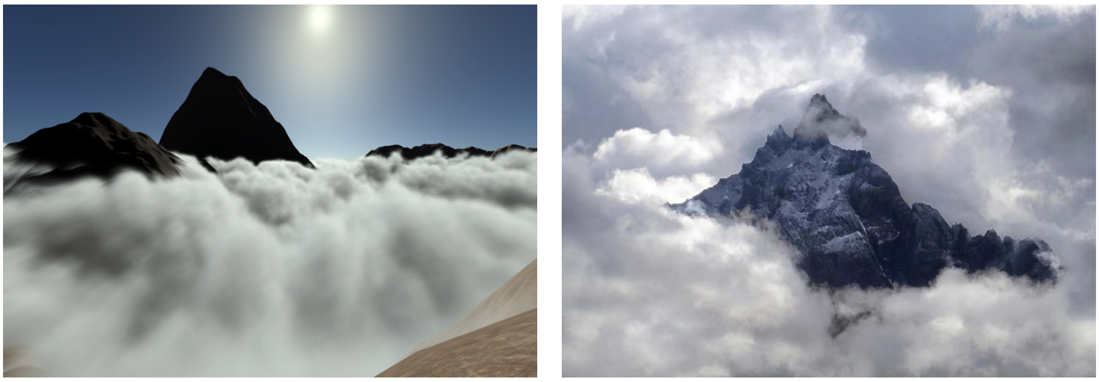
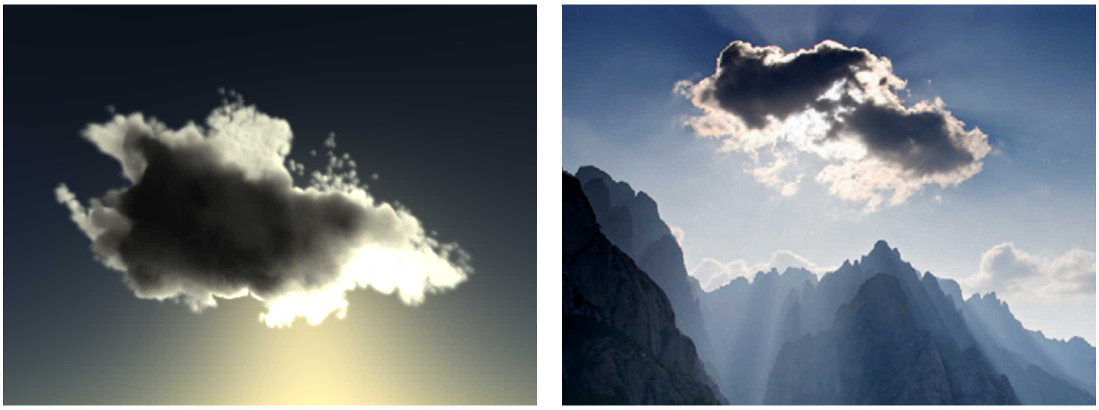


Figure 9.3: Thick cloud layer fully encompassing a mountain peak from above with high particle opacity.



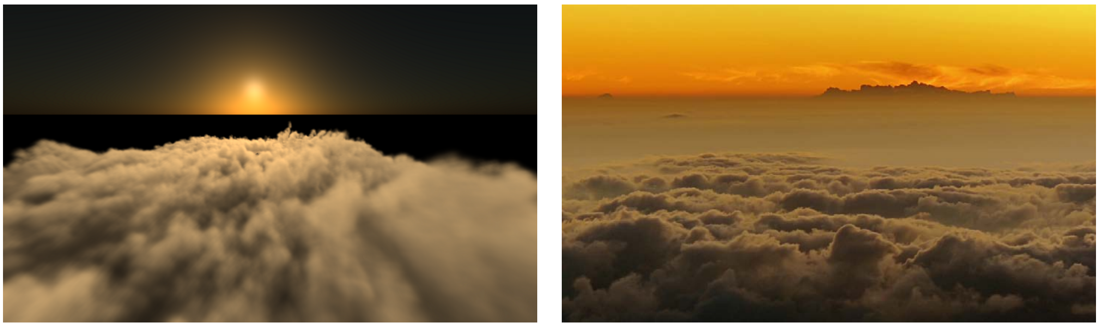
(a) Our result. (b) Reference photograph [Bir13].

Figure 9.4: Comparison of our method with reference photography. Clouds are flowing through a mountainous terrain using the LBM and STLP methods.



(a) Our result. (b) Reference photograph [Pru15].

Figure 9.5: Comparison of our cloud rendering method with a reference photograph. The cloud was rendered with high self-shadowing multiplier and the Henyey-Greenstein phase function.



(a) Our results.

(b) Reference photograph [Ker09]

Figure 9.6: Layer of clouds generated with our method compared with a reference photograph. The image also showcases how the sun's color is changed dynamically based on the sun's elevation.

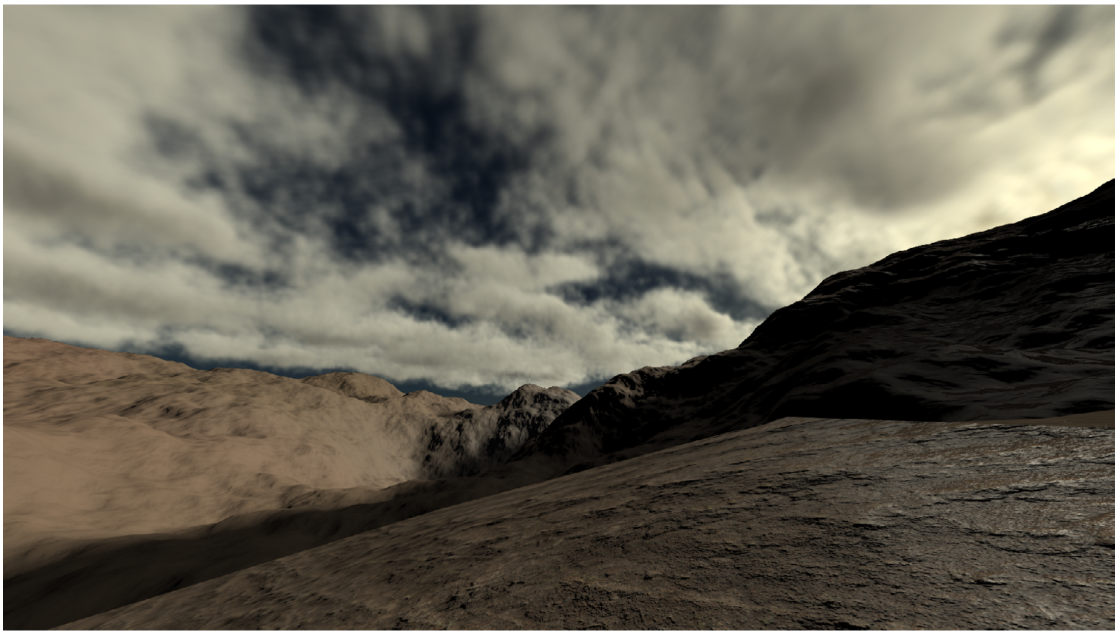


Figure 9.7: Clouds flowing above a mountain range captured from below. Both STLP and LBM were used for generating the cloudscape.

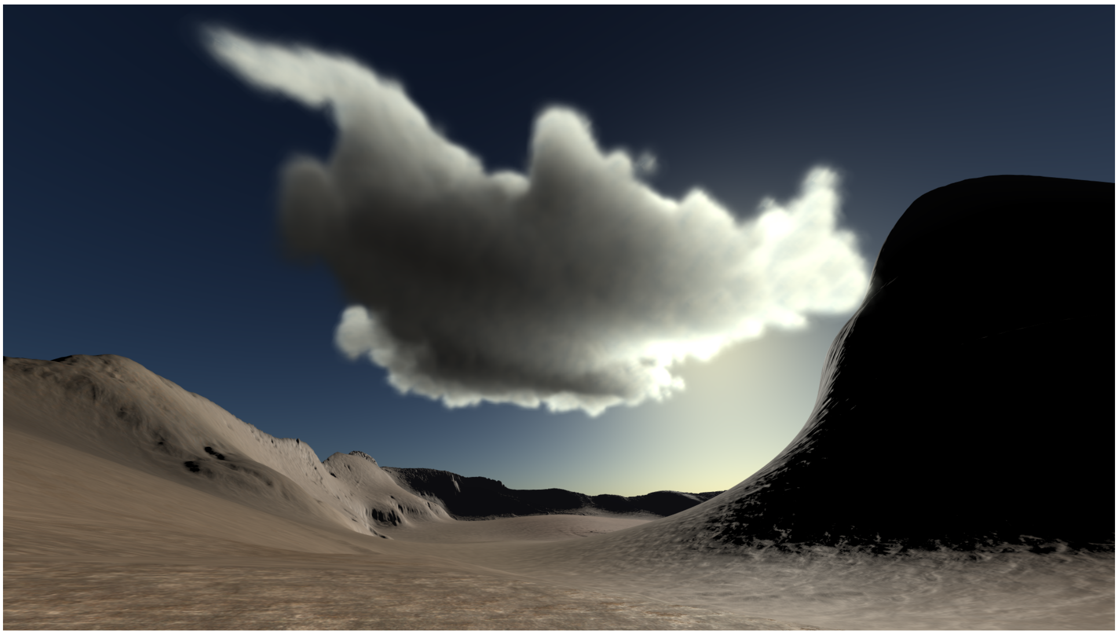
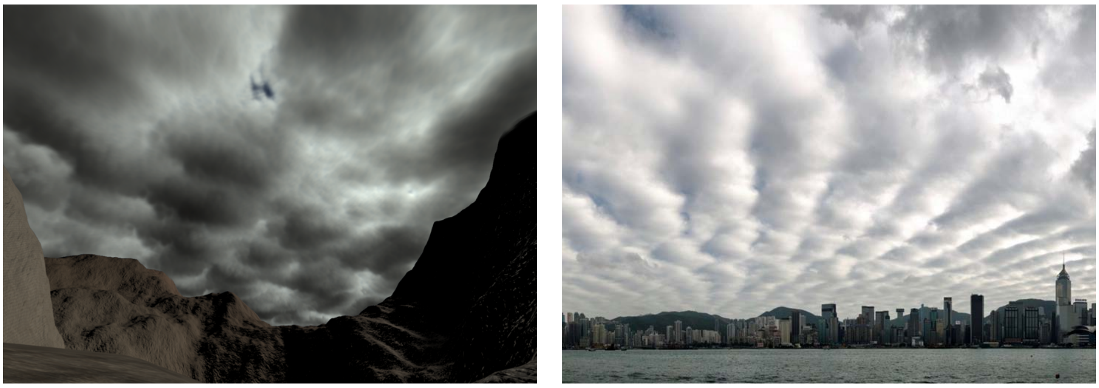


Figure 9.8: Large cloud generated from box with simulation of LBM and STLP applied for short amount of time.



(a) Our result.

(b) Reference photograph [wmoa]

Figure 9.9: Comparison of generated stratocumulus clouds with a reference photograph.

9.1 Measurements

The algorithm was tested on two computers: a desktop and a notebook. See Table 9.1 for their specifications.

	Desktop	Notebook
Operating system	MS Windows 10 Home (64-bit)	MS Windows 10 Home (64-bit)
CPU	Intel Core i7-6700K @ 4.00GHz 4.00GHz	Intel Core i7-7700HQ @ 2.80GHz 2.81GHz
GPU	NVIDIA GeForce GTX 1080	NVIDIA GeForce GTX 1050 Ti
Memory	32.0 GB RAM	16.0 GB RAM
Compute capability	6.1	6.1
CUDA driver version	10.0	10.0
Streaming multiprocessors	20	6
CUDA cores	2560	768

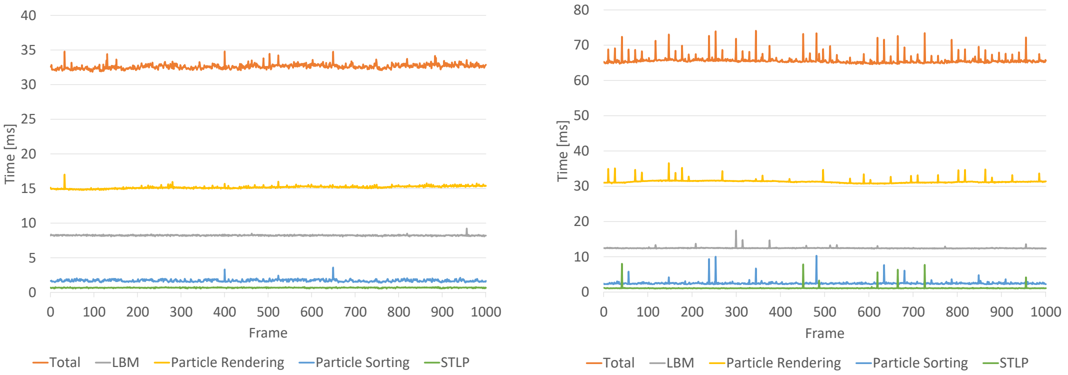
Table 9.1: Specifications of computers that were used for measurements.

For time measurements in the framework, we provide an extensive timer system that tracks the most important and costly parts of the simulation process. Timers can be added in code by using the `TimerManager::createTimer()` function. Each timer will be automatically added to the UI and its measured frame times will be appended to the generated benchmark `.csv` file.

We mention this system because the application and its speed are mainly dependent on the configuration of our cloud rendering algorithm. Attributes such as point size, particle opacity, sprite texture selection or particle shadow alpha have great impact on the final speed of the application. Generally, the larger and more opaque the particles, the slower the rendering process. The main cause of this is pixel overdraw where each particle draws multiple pixels to the screen. Furthermore, the closer the camera gets to a large group of particles, the larger slowdown will occur due to said overdraw. Complexity of the cloud rendering algorithm also depends on the number of slices that are used. The higher the slice count, the more computationally expensive the algorithm becomes. 256 slices were used for all the presented result images and measurements. The application was run in fullscreen with full HD resolution and 4xMSAA during all tests.

In the first shown pair of diagrams (Figure 9.10) we present a benchmark of our application in its default settings on both the desktop computer and the notebook. The lattice size is set to $100 \times 60 \times 100$ (width $\times$ height $\times$ depth) and 500k particles are drawn. Furthermore, particles that are below CCL are not drawn since they should not be visible during the simulation. As the reader can observe, the application holds a steady real-time frame rate with these settings maintaining around 30 frames per second (FPS) or drawing each frame in 32.61ms on average on the desktop. On the notebook, interactive times are achieved with 65.60ms needed to draw and simulate one frame on average. During the measurement we have used a fixed camera as shown in Figure 9.11. Average times for all the measured systems are shown in Table 9.2.

The application and all its systems show stable frame rate when fixed camera is used. As the reader can notice, the most costly part of our system is the rendering by a large margin. On the other hand, the STLP simulation has negligible impact on the performance.



(a) Desktop benchmark.

(b) Notebook benchmark.

Figure 9.10: Benchmark of our application with 500k active particles and LBM with $100 \times 60 \times 100$ lattice nodes. Total time shown denotes actual time needed to generate one frame including other operations such as drawing terrain etc.

	Total	LBM	Rendering	Sorting	STLP
Desktop	32.61	8.24	15.17	1.70	0.68
Notebook	65.60	12.46	31.30	2.48	1.12

Table 9.2: Average times [ms] from the benchmark presented in Figure 9.10

In a second benchmark (see Figure 9.12), more extreme conditions were tested with 10 million active particles and a lattice with resolution 100^3. Furthermore, the camera was moved around the scene which led to peaks and valleys in frame rate due to rapid changes in pixel overdraw as described earlier. Nonetheless, running all the systems, the application averaged 6.30 frames per second or 158.62ms per frame which is in our opinion adequate. Note that all images of our results presented in this thesis use maximally one million active particles (500k particles in most cases). Beyond particle rendering, all other operations, including LBM, show stable frame rates once again. For average times for this benchmark, please see Table 9.3.

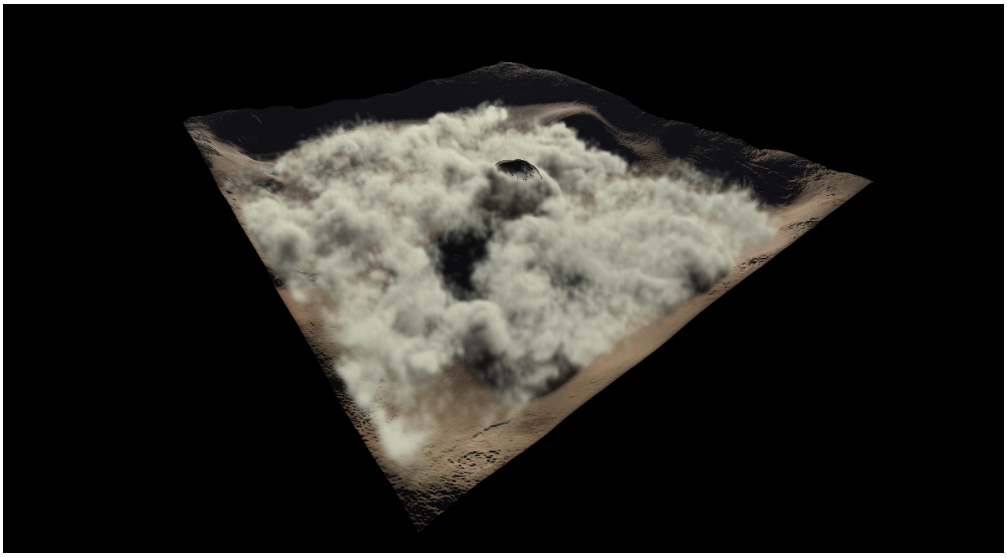


Figure 9.11: Cropped screenshot obtained during the benchmarking process using a fixed camera.

Total	LBM	Rendering	Sorting	STLP
158.62	31.98	94.49	15.63	8.65

Table 9.3: Average times [ms] from the benchmark presented in Figure 9.12

Additionally to the benchmarks, in Table 9.4 we present how the performance of our LBM implementation scales with lattice size and particle count. Lastly, in Table 9.5, the speed of the STLP simulation is presented. As the reader can notice, the STLP simulation can be run for large amounts of particles quite easily.

9.1.1 CPU and GPU Comparison

For more in-depth measurements of our older LBM implementation we would like to refer the reader to our standalone project report available at `www.martincap.io/ProjectFuji/LatticeBoltzmann_report.pdf` where CPU and GPU implementations of the LBM are compared. For CPU measurements of the STLP simulator, please see Duarte's thesis [Dua16] where he presents his results using a single-threaded CPU implementation. During the creation of this thesis, we have shortly tested our CPU implementation of the STLP simulator and obtained similar results as Duarte.

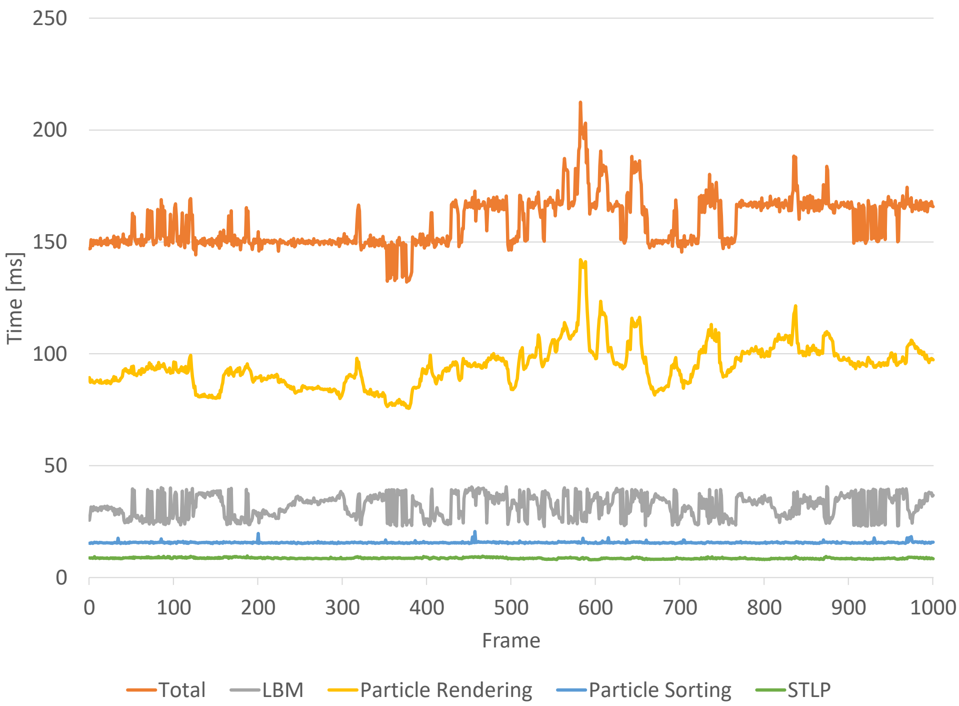


Figure 9.12: Benchmark of our application on the desktop with 10 million particles and LBM with 100^3 lattice nodes. During the benchmarking the camera was purposefully moved around resulting in fluctuations in particle rendering speed.

Particles / Lattice	500k	1m	5m	10m
50^3	2.56	3.17	8.03	14.23
100^3	13.73	14.27	18.74	24.91
150^3	44.16	44.87	50.98	59.10
200^3	103.82	106.18	123.04	127.79

Table 9.4: Table showing simulation frame time [ms] for LBM with relation to lattice resolution and number of particles. Values are averaged from 1000 frames.

Particles	500k	1m	5m	10m	50m
Time [ms]	0.94	1.28	4.04	7.63	35.50

Table 9.5: Table showing average time [ms] needed for one STLP simulation step with relation to number of particles. Values are averaged from 1000 frames.

10 Conclusion

In this thesis we have demonstrated how the SkewT/LogP cloud simulation method proposed by Duarte [Dua16] can be coupled with the Lattice Boltzmann method to produce orographic clouds. An extensive framework was implemented using C++, OpenGL and CUDA. Majority of the framework takes advantage of the GPU and exploits the parallelizability of the above-mentioned methods. Additionally, a volume rendering algorithm based on Green's paper [Gre08] was implemented. The algorithm was extended with anisotropic light scattering by using phase functions to increase intensity of less occluded cloud particles. Everything put together runs in real-time including other implemented algorithms such as exponential variance shadow maps, mesh instancing, or PBR shaders for example. Furthermore, a user-friendly graphic interface was created for the application. With it, users can draw cloud particles on the terrain, modify the SkewT/LogP diagram and much more.

We feel that this approach to cloud simulation would require a lot more work and attention to be useful in practical scenarios. Currently, it is heavily dependent on user experimentation where multitude of its parameters can have great impact on the generated cloud shapes. For this purpose, we provide a wide range of tools that make the experimentation process with both simulation methods as easy as possible. However, one could say that Duarte's approach clashes with LBM in many ways. The SkewT/LogP simulator requires prepared emitters on the ground with time-varying variables to feed the system with interesting cloud profiles. On the other hand, LBM is a physically-based approach that, in its current state, does not give enough options to determine the final flow of its simulated fluid. This results in particle clusters being separated and creating more viscous fluidlike results than desired. These issues are easily spotted with our cloud visualization technique which does not account for lone particles, rendering light dots in the sky where the density of the condensed particles is too low to be visible by the naked eye.

With these issues in mind, the method is most suitable for cloudscapes with high cloud coverage. Additionally, as suggested in the previous chapter, with some experimentation and with our set of tools, visually pleasing results with interesting cloud formations are easily obtainable. In brief, we have accomplished the goals we set out for this thesis. Beyond that, a great deal of time was spent on adding a cloud visualization which was not initially planned. Thanks to this, we have created a comprehensive cloud simulator that may be easily extended in the future.

11 Future Work

There are countless areas where one could take our work and improve upon it. As mentioned earlier, parametrization of the LBM, from which the whole simulation process would greatly benefit, is not implemented in our framework. Another area of improvement would be addition of more sophisticated emitters whose shape and convective temperature profiles could be adjusted using time dependent curves or functions. To take it even further, a surface heating and cooling algorithm that would act as an emitter would be another major improvement to the framework. Inclusion of Hošek's solar radiance model [HW13] in the sky model would give us a realistic sun appearance in place of our current simplified visualization. Last but not least, an ability to render physically realistic night scenes should also be considered in future updates of our framework.

Bibliography

[Ahr11] C. Donald Ahrens. *Essentials of Meteorology: An Invitation to the Atmosphere*. Cengage Learning, 2011.

[ams12] Pseudoadiabatic lapse rate. `http://glossary.ametsoc.org/wiki/Pseudoadiabatic_lapse_rate`, 2012. (Online; accessed: 05/05/2019).

[And10] David G. Andrews. *An Introduction to Atmospheric Physics*. Cambridge University Press, 2010.

[And17] Ben Anderson. Hošek-Wilkie sky simulator. `https://github.com/benanders/Hosek-Wilkie`, 2017. (Online; accessed: 04/07/2019).

[BBH13] Tomáš Barák, Jiří Bittner, and Vlastimil Havran. Temporally coherent adaptive sampling for imperfect shadow maps. In *Computer Graphics Forum*, volume 32, pages 87–96. Wiley Online Library, 2013.

[Bir13] Bill Birtwhistle. The mountain in the clouds. `https://www.flickr.com/photos/12496504@N06/8520510333`, 2013. (Online; accessed: 05/20/2019).

[BM11] Yuanxun Bill Bao and Justin Meskas. Lattice Boltzmann method for fluid simulations. *Department of Mathematics, Courant Institute of Mathematical Sciences, New York University*, 2011.

[BN04] Antoine Bouthors and Fabrice Neyret. Modeling Clouds Shape. In M. Alexa and E. Galin, editors, *Eurographics (Short Presentations)*. Eurographics Association, 2004.

[BNM$^+$08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 173–182, New York, USA, 2008. ACM.

[Bol80] David Bolton. The computation of equivalent potential temperature. *Monthly Weather Review*, 108(7):1046–1053, 1980.

[Bou88] Paul Bourke. Points, lines, and planes. `http://paulbourke.net/geometry/pointlineplane/`, 1988. (Online; accessed: 12/16/2018).

[BS13] Atoossa Bakhshaii and Roland Stull. Saturated pseudoadiabats—A non-iterative approximation. *Journal of Applied Meteorology and Climatology*, 52:5–15, 01 2013.

[CS92] William M. Cornette and Joseph G. Shanks. Physically reasonable analytic expression for the single-scattering phase function. *Applied Optics*, 31(16):3152–3160, 1992.

[Daw18] Paul Dawkins. Calculus III: Interpretations of partial derivatives. `http://tutorial.math.lamar.edu/Classes/CalcIII/PartialDerivInterp.aspx`, 2018. (Online; accessed: 04/29/2019).

[DIO$^+$12] Yoshinori Dobashi, Wataru Iwasaki, Ayumi Ono, Tsuyoshi Yamamoto, Yonghao Yue, and Tomoyuki Nishita. An inverse problem approach for automatically adjusting the parameters for rendering clouds using photographs. *ACM Transactions on Graphics*, 31(6):145–1, 2012.

[DIYN17] Yoshinori Dobashi, Kei Iwasaki, Yonghao Yue, and Tomoyuki Nishita. Visual simulation of clouds. *Visual Informatics*, 1(1):1–8, 2017.

[DKY$^+$00] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Nishita. A simple, efficient method for realistic animation of clouds. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 19–28. ACM Press/Addison-Wesley Publishing Co., 2000.

[DL06] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 161–165. ACM, 2006.

[DNO98] Yoshinori Dobashi, Tomoyuki Nishita, and Tsuyoshi Okita. Animation of clouds using cellular automaton. In *Proceedings of Computer Graphics and Imaging*, volume 98, pages 251–256, 1998.

[DNYO98] Yoshinori Dobashi, Tomoyuki Nishita, Hideo Yamashita, and Tsuyoshi Okita. Modeling of clouds from satellite images using metaballs. In *Proceedings Pacific Graphics' 98. Sixth Pacific Conference on Computer Graphics and Applications*, pages 53–60. IEEE, 1998.

[DSY10] Yoshinori Dobashi, Yusuke Shinzo, and Tsuyoshi Yamamoto. Modeling of clouds from a single photograph. In *Computer Graphics Forum*, volume 29, pages 2083–2090. Wiley Online Library, 2010.

[Dua16] Rui Pedro Monteiro Amaro Duarte. *Realistic Simulation and Animation of Clouds using SkewT/LogP Diagrams.* PhD thesis, Universidade da Beira Interior, 2016.

[dVa] Joey de Vries. Lighting. `https://learnopengl.com/PBR/Lighting`. (Online; accessed: 01/12/2019).

[dVb] Joey de Vries. Text rendering. `https://learnopengl.com/In-Practice/Text-Rendering`. (Online; accessed: 01/12/2019).

[DW16] Ralf Deiterding and Stephen L. Wood. Predictive wind turbine simulation with an adaptive Lattice Boltzmann method for moving boundaries. In *Journal of Physics: Conference Series*, volume 753, pages 1–11. IOP Publishing, 2016.

[DYN06] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. A controllable method for animation of earth-scale clouds. *Proceedings of Computer Animation and Social Agents*, pages 43–52, 2006.

[EMP$^+$03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, William R. Mark, and John C. Hart. Texturing and modeling:A procedural approach. 2003.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 15–22. ACM, 2001.

[Gre08] Simon Green. Volumetric particle shadows. *NVIDIA Developer Zone*, 2008.

[GRWS04] Robert Geist, Karl Rasche, James Westall, and Robert Schalkoff. Lattice-Boltzmann lighting. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, pages 355–362, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[GSW07] Robert Geist, Jay Steele, and James Westall. Convective clouds. In *Proceedings of the Third Eurographics Workshop on Natural Phenomena*, NPH'07, pages 23–30, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[Häg18] Fredrik Häggström. Real-time rendering of volumetric clouds. Master's thesis, Umeå University, 2018.

[Har03] Mark J. Harris. *Real-Time Cloud Simulation and Rendering.* PhD thesis, University of North Carolina at Chapel Hill, 2003.

[HBSL03] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics H]ardware*, pages 92–101. Eurographics Association, 2003.

[Hil16] Sebastien Hillaire. Physically based sky, atmosphere and cloud rendering in Frostbite. In *ACM SIGGRAPH 2016 Course: Physically Based Shading in Theory and Practice*, 2016.

[HL01] Mark J. Harris and Anselmo Lastra. Real-time cloud rendering. In *Computer Graphics Forum*, volume 20, pages 76–85. Wiley Online Library, 2001.

[HW12] Lukáš Hošek and Alexander Wilkie. An analytic model for full spectral sky-dome radiance. *ACM Transactions on Graphics*, 31(4):95, 2012.

[HW13] Lukáš Hošek and Alexander Wilkie. Adding a solar-radiance function to the Hošek-Wilkie skylight model. *IEEE Computer Graphics and Applications*, 33(3):44–52, 2013.

[Ish] Miaki Ishii. Normal modes and constraints on mantle 1D and 3D structure. `https://seismo.berkeley.edu/wiki_cider/images/d/df/Miaki_ishii_cider2016.pdf`. (Online; accessed: 01/05/2019).

[Kat75] George W. Kattawar. A three-parameter analytic phase function for multiple scattering calculations. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 15(9):839–849, 1975.

[Ker09] Tom Kerr. Above the cloud layer. `http://apacificview.blogspot.com/2009/11/above-cloud-layer.html`, 2009. (Online; accessed: 05/12/2019).

[KPH$^+$03] Joe Kniss, Simon Premože, Charles Hansen, Peter Shirley, and Allen McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[Lat07] Jonas Latt. How to implement your DdQq dynamics with only q variables per node (instead of 2q). Technical report, 2007.

[LM08] Andrew Lauritzen and Michael McCool. Layered variance shadow maps. In *Proceedings of Graphics Interface 2008*, pages 139–146. Canadian Information Processing Society, 2008.

[Maq17] Nicolas Maquignon. Everything you need to know about the Lattice Boltzmann method. `http://feaforall.com/creating-cfd-solver-lattice-boltzmann-method/`, 2017. (Online; accessed: 12/30/2018).

[MDN02] Ryo Miyazaki, Yoshinori Dobashi, and Tomoyuki Nishita. Simulation of cumuliform clouds based on computational fluid dynamics. *Proceedings Eurographics 2002 (Short Presentation)*, pages 405–410, 2002.

[Met17] Micha Mettke. Nuklear. `https://github.com/vurtun/nuklear`, 2017. (Online; accessed: 11/18/2018).

[MF$^+$04] John H. Mathews, Kurtis D. Fink, et al. *Numerical Methods Using MATLAB*, volume 3. Pearson Prentice Hall Upper Saddle River, NJ, 2004.

[MS17] Nadya Moisseeva and Roland Stull. Technical note: A noniterative approach to modelling moist thermodynamics. *Atmospheric Chemistry and Physics*, 17:15037–15043, 12 2017.

[MYDN01] Ryo Miyazaki, Satoru Yoshida, Yoshinori Dobashi, and Tomoyuki Nishita. A method for modeling clouds based on atmospheric fluid dynamics. In *Proceedings Ninth Pacific Conference on Computer Graphics and Applications. Pacific Graphics 2001*, pages 363–372. IEEE, 2001.

[Nav07] Mirko Navara. Pravděpodobnost a matematická statistika. *Skriptum ČVUT, Praha*, 2007.

[NR92] Kai Nagel and Ehrhard Raschke. Self-organizing criticality in cloud formation. *Physica A: Statistical Mechanics and its Applications*, 182(4):519–531, 1992.

[Per02a] Ken Perlin. Improved noise reference implementation. `https://mrl.nyu.edu/~perlin/noise/`, 2002. (Online; accessed: 2019-04-14).

[Per02b] Ken Perlin. Improving noise. In *ACM Transactions on Graphics*, volume 21, pages 681–682. ACM, 2002.

[PH89] Ken Perlin and Eric M. Hoffert. Hypertexture. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 253–262. ACM, 1989.

[Pre03] Simon Premože. Light transport in participating media. *ACM SIGGRAPH Course: Light and Color in the Outdoors*, 2003.

[Pru15] Jean-Colas Prunier. Simulating the colors of the sky. `https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/simulating-sky/simulating-colors-of-the-sky`, 2015. (Online; accessed: 05/12/2019).

[PSM93] Richard Perez, Robert Seals, and Joseph Michalsky. All-weather model for sky luminance distribution—Preliminary configuration and validation. *Solar Energy*, 50(3):235–245, 1993.

[PSS99] A. J. Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 91–100, New York, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Py16] Yeung Pui-yi. Waterfall-like orographic clouds. `http://www.weather.gov.hk/education/article_e.htm?title=ele_00484`, 2016. (Online; accessed: 01/14/2019).

[qui04] A quick derivation relating altitude to air pressure. `http://archive.psas.pdx.edu/RocketScience/PressureAltitude_Derived.pdf`, 2004. (Online; accessed: 04/12/2019).

[Rá10] Daniel Rákos. Efficient gaussian blur with linear sampling. `http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/`, 2010. (Online; accessed: 12/20/2018).

[Sch] Jeff Schmaltz. Wave clouds near amsterdam island. `https://earthobservatory.nasa.gov/images/6151/wave-clouds-near-amsterdam-island`. (Online; accessed: 01/11/2019).

[SN10] Martin Schreiber and Philipp Neumann. *GPU Based Simulation and Visualization of Fluids with Free Surfaces.* PhD thesis, Technische Universität München, 2010.

[Sta99] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, New York, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[SV15] Andrew Schneider and Nathan Vos. The real-time volumetric cloudscapes of Horizon: Zero Dawn. *ACM SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2015.

[SV17] Andrew Schneider and Nathan Vos. Nubis, authoring real-time volumetric cloudscapes with the Decima Engine. *ACM SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2017.

[Wal] Simon Walton. Finding the intersection point of two line segments in $\mathbb{R}^2$. `http://www.cs.swan.ac.uk/~cssimon/line_intersection.html`. (Online; accessed: 02/18/2019).

[Wan03] Niniane Wang. Realistic and fast cloud rendering in computer games. In *ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1. Citeseer, 2003.

[Wan04] Niniane Wang. Realistic and fast cloud rendering. *Journal of Graphics Tools*, 9(3):21–40, 2004.

[WBSP18] Mark A. Woodgate, George N. Barakos, Rene Steijl, and Gavin J. Pringle. Parallel performance for a real time Lattice Boltzmann code. *Computers & Fluids*, 173:237–258, 2018.

[wmoa] Classifying clouds. `https://public.wmo.int/en/WorldMetDay2017/classifying-clouds`. (Online; accessed: 05/15/2019).

[wmob] Orographic influence on the leeward side. `https://cloudatlas.wmo.int/orographic-influence-on-the-leeward-side.html`. (Online; accessed: 01/14/2019).

[wmoc] Orographic influence on the windward side. `https://cloudatlas.wmo.int/orographic-influences-on-the-windward-side.html`. (Online; accessed: 01/14/2019).

[wmod] Orographic influences on clouds: Introduction. `https://cloudatlas.wmo.int/orographic-influences-on-clouds-introduction.html`. (Online; accessed: 01/14/2019).

[Woo16] Stephen Lloyd Wood. *Lattice Boltzmann Methods for Wind Energy Analysis.* PhD thesis, University of Tennessee, 2016.

[WZF$^+$03] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman. Blowing in the wind. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 75–85. Eurographics Association, 2003.

[Yus14] Egor Yusov. High-performance rendering of realistic cumulus clouds using pre-computed lighting. In *High Performance Graphics*, pages 127–136, 2014.

A Obtaining Relation Between T and P Under Adiabatic Changes

Once the air parcel has been lifted to level (height) z, it has the same pressure as the environment, that is, $P_e(z) = P_p(z)$. Let $a = P_e(z) = P_p(z)$. By substituting a in Equation 2.2 (ideal gas law, $P = \rho RT$), we obtain $a = \rho_e(z)RT_e(z)$ and $a = \rho_p(z)RT_p(z)$. These can be rewritten to

$$\rho_e(z) = \frac{a}{RT_e(z)} \tag{A.1}$$

and

$$\rho_p(z) = \frac{a}{RT_p(z)} \tag{A.2}$$

By substituting $(dP_e/dz)_z$ with $-\rho_e(z)g$ (using hydrostatic balance, see Equation 2.3) in Equation 2.4 we get

$$\rho_p(z)\frac{dv}{dt} = -(-\rho_e(z)g) - \rho_p(z)g \tag{A.3}$$

By substituting values of $\rho_e(z)$ and $\rho_p(z)$ in Equation A.3 we obtain

$$\frac{a}{RT_p(z)}\frac{dv}{dt} = \frac{ag}{RT_e(z)} - \frac{ag}{RT_p(z)} \tag{A.4}$$

By dividing the equation with a and simplifying we get (in steps)

$$\frac{dv}{dt} = \frac{RT_p(z)g}{RT_e(z)} - \frac{RT_p(z)g}{RT_p(z)} \tag{A.5}$$

$$\frac{dv}{dt} = g\frac{(T_p(z))^2 - T_p(z)T_e(z)}{T_e(z)T_p(z)} \tag{A.6}$$

$$\frac{dv}{dt} = g\frac{T_p(z)(T_p(z) - T_e(z))}{T_e(z)T_p(z)} \tag{A.7}$$

$$\frac{dv}{dt} = g\frac{T_p(z) - T_e(z)}{T_e(z)} \tag{A.8}$$

B Exponential Variance Shadow Maps

For rendering shadows in our framework, the exponential variance shadow maps (EVSM) method proposed by Lauritzen and McCool [LM08] was implemented. The method is an extension of variance shadow maps (VSM) proposed by Donnelly and Lauritzen [DL06].

Basic idea of VSM lies in the usage of Chebyshev's inequality. Instead of storing depths in the depth map, the first and second moments $M_1 = x$ and $M_2 = x^2$ of depth x are stored instead. From these moments, we can compute the mean $\mu = E(x) = M_1$ and the variance $\sigma^2 = E(x^2) - E(x)^2 = M_2 - M_1^2$. The Chebyshev's inequality then describes the upper bound of shadow intensity p_{max} for a given fragment at depth t as

$$P(x > t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{B.1}$$

The main advantage of using this approach lies in the fact that blurring the depth map and applying other filters such as mipmapping or anisotropic filtering are sensible operations. On the other hand, this does not hold for the naive shadow mapping technique. Thanks to this, generating high quality soft shadows is possible by blurring the depth map texture. However, one large disadvantage of this method is that it is prone to light bleeding because only the upper bound p_{max} of the shadow intensity is computed. This leads to unwanted and unnaturally lit areas, especially in penumbras of shadows that are cast onto different shadows as shown in Figure B.1. This can be partly resolved by clamping p_{max} by some minimum value which reduces light bleeding but also reduces the quality of the shadows when set too high.

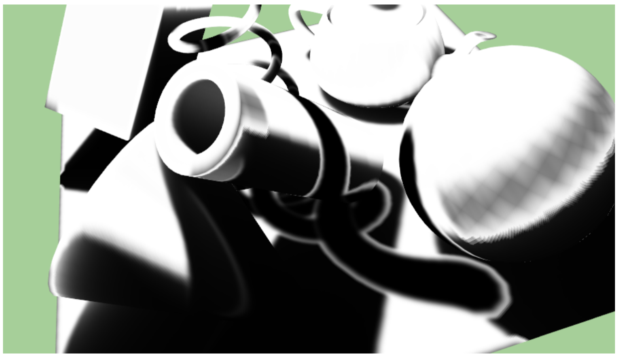
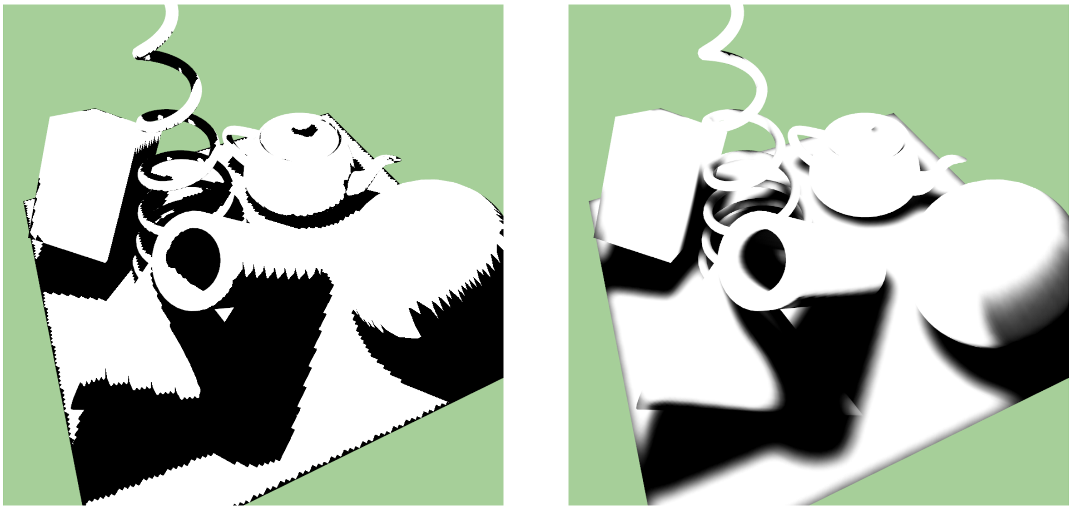


Figure B.1: Very apparent example of light bleeding that was produces with depth map blurring.

To solve the light bleeding issue, Lauritzen and McCool propose using the exponential warping function. More specifically, the saved values to the depth map should be warped using a function e^{cx} where c is a user-defined constant and x is the fragment depth. Therefore, instead of storing the first and second moments of the depth in the first pass, we store e^{cx} and $(e^{cx})^2$ instead. The same warping is applied in the second pass when drawing the scene from camera's point of view. The warped values are then used in the Chebyshev's inequality to compute the shadow intensity of a given fragment.

Gaussian Blur. Additionally, optimized two-pass gaussian blur with linear sampling as described by Rákos [Rá10] was implemented as part of the project. Thus, creation of high quality soft shadows with low depth map resolutions is possible as shown in Figure B.2.

More Information. For more information about the implemented method, please see my personal website: `www.martincap.io`.



(a) Naive shadows.

(b) EVSM shadows.

Figure B.2: Comparison of naive shadow mapping technique and EVSM with optimized 9x9 gaussian blur. Resolution of 256x256 depth map resolution.

C Subgrid Model of LBM

One of the main purposes of this work is the use of LBM in orographic cloud simulation. For that reason, we want to reach high Reynolds number (Re) flows with our simulation. The higher the value of Re, the more turbulent results we get, possibly creating eddy currents in proximity to colliders/obstacles. Multiple methods exist for achieving these desired effects. Let us describe the model as presented by Wei et al. in [WZF$^+$03].

Smagorinsky Subgrid Model

At unit time step, the fluid kinematic viscosity ν is related to the relaxation time τ by

$$\nu = \frac{2\tau - 1}{6} \tag{C.1}$$

Since we want to reach high Reynolds number (Re) flows, we want the value of ν to be as small as possible, meaning that we send $\tau \to 1/2$. Numerical issues can arise when $\tau \to 1/2$, leading to instability.

This problem can be partly resolved using the subgrid model [WZF$^+$03]. As described in the paper, the roots of the numerical instability lie in the inability of the LBM to represent flow dynamics on scales smaller than the lattice spacing. Since the lattice may be coarse, we lack the ability to transfer energy in smaller scale, making us unable to stabilize the model. The Smagorinsky subgrid model is used to alleviate these issues to some degree. It is important to note that this model does not in fact subdivide the lattice in any manner, it just updates the collision step computation to capture these small energy transfers. In the Smagorinsky subgrid model, the value of τ is allowed to vary over lattice and change during each frame. This method also does not break the parallelizability of the application, since we only access values of the neighbouring nodes.

To compute τ_{new} for a given node, we must calculate the magnitude of the local strain tensor S. For this, we use equations provided in the dissertation thesis by Wood [Woo16]. First, let us define a non-equilibrium distribution as

$$f_i^{neq} = f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t) \tag{C.2}$$

The local strain tensor can be then obtained using the set of equations (note that we use different indexation of our D3Q19 model than in [Woo16]):

$$s_{xx} = \alpha(f_3^{neq} + f_4^{neq} + f_7^{neq} + f_8^{neq} + f_9^{neq} + f_{10}^{neq} + f_{15}^{neq} + f_{16}^{neq} + f_{17}^{neq} + f_{18}^{neq}) \tag{C.3}$$

$$s_{xy} = s_{yx} = \alpha(f_{15}^{neq} + f_{16}^{neq} + f_{17}^{neq} + f_{18}^{neq}) \tag{C.4}$$

$$s_{xz} = s_{zx} = \alpha(f_7^{neq} - f_8^{neq} + f_9^{neq} - f_{10}^{neq}) \tag{C.5}$$

$$s_{yy} = \alpha(f_5^{neq} + f_6^{neq} + f_{11}^{neq} + f_{12}^{neq} + f_{13}^{neq} + f_{14}^{neq} + f_{15}^{neq} + f_{16}^{neq} + f_{17}^{neq} + f_{18}^{neq}) \tag{C.6}$$

$$s_{yz} = s_{zy} = \alpha(-f_{11}^{neq} + f_{12}^{neq} + f_{13}^{neq} - f_{14}^{neq}) \tag{C.7}$$

$$s_{zz} = \alpha(f_1^{neq} + f_2^{neq} + f_7^{neq} + f_8^{neq} + f_9^{neq} + f_{10}^{neq} + f_{11}^{neq} + f_{12}^{neq} + f_{13}^{neq} + f_{14}^{neq}) \tag{C.8}$$

where c_s is the physical speed of sound of the fluid and it is related to c as $c_s = c/\sqrt{3}$, ρ is the macroscopic density and

$$\alpha = \frac{-1}{2\rho c_s^2 \tau} \tag{C.9}$$

The magnitude of the local strain tensor is

$$|S| = \sqrt{2(s_{xx}^2 + s_{yy}^2 + s_{zz}^2 + 2(s_{xy}^2 + s_{xz}^2 + s_{yz}^2))} \tag{C.10}$$

The Smagorinsky eddy viscosity, given by

$$\nu_t = C^2|S| \tag{C.11}$$

is then added to the shear viscosity to locally adjust the value of τ as

$$\tau_{new} = 3(\nu + C^2|S|) + 1/2) \tag{C.12}$$

where C is the Smagorinsky constant that typically has the value of 0.3.

D Physics and Thermodynamics Basics

In this appendix, we would like to provide a very short glossary of terms that were not explained in the main text. The definitions are taken and paraphrased from [Ahr11] and `www.engineeringtoolbox.com`.

Pressure P Units. SI unit of pressure is 1 pascal [Pa] which is 1 N/m^2. Bar is another commonly used unit of pressure that is not approved as part of the SI unit system. It holds that

$$1\,\text{bar} = 10^5\,\text{Pa} = 1000\,\text{hPa} = 100\,\text{kPa} \tag{D.1}$$

or as is commonly used

$$1\,\text{millibar} = 1\,\text{hPa} \tag{D.2}$$

Temperature T. Temperature is the measure of the average kinetic energy of each individual molecule in a substance:

$$KE_{avg} = \frac{3}{2}k \cdot T \tag{D.3}$$

where KE_{avg} is the average kinetic energy of molecule and $k = 1.381 \cdot 10^{-23}$ is the Boltzmann constant.

Multiple scales/units are used to measure temperature, in this thesis only degree Celsius [°C] and degree Kelvin [K] are used. Celsius and Kelvin share the same incremental scale and one unit of Kelvin is equal in size to one unit of Celsius. To calculate temperature in degree Kelvin from degree Celsius, add 273.15 to Celsius as follows:

$$T(\text{K}) = T(^\circ\text{C}) + 273.15 \tag{D.4}$$

Dry Bulb Temperature. Dry bulb temperature is the most commonly used air property usually referred to as “air temperature”. It basically corresponds to ambient air temperature. The dry bulb temperature is obtained with thermometers that are not affected by moisture of the air.

Wet Bulb Temperature Wet bulb temperature is the adiabatic saturation temperature and can be measured by using a thermometer with the bulb wrapped in wet muslin (a lightweight cotton cloth).

Dew Point Temperature. Dew point temperature is a temperature where water vapor starts to condense out of the air and therefore becomes completely saturated. The measurement of the dew point is related to humidity. If the dew point temperature is close to the dry air temperature, the relative humidity is high, otherwise, it is low.

Lapse Rate. The rate at which air temperature in Earth's atmosphere decreases with height. It is the negative of the rate the temperature changes with altitude:

$$\Gamma = -\frac{dT}{dz} \tag{D.5}$$

The average lapse rate in the atmosphere up to 11km is about 6.5 degrees Celsius for every 1000 meters.

Buoyancy. Buoyancy, or upthrust, is the tendency of a body to float or rise when submerged in a fluid. The resultant upward force acting on the object that is immersed in the fluid is called the buoyant force and can be expressed as

$$F = V\gamma = V\rho g \tag{D.6}$$

where F [N] is the buoyant force, V [m^3] is volume of the body, $\gamma = \rho g$ [N/m^3] is the specific weight of fluid, ρ [kg/m^3] is the density of fluid and $g = 9.81$ [m/s^2] is the acceleration of gravity.

Thermal Energy. Thermal energy is a kinetic energy of all the molecules in a system added together as opposed to temperature, which is a measure of the average kinetic energy for each molecule.

$$U = \frac{3}{2} \cdot N \cdot k \cdot T \tag{D.7}$$

where U is the thermal energy of ideal gas, N is the number of molecules, $k = 1.381 \cdot 10^{-23}$ is the Boltzmann constant and T is temperature.

Heat. Heat is the amount of thermal energy added to or removed from a system. It is an energy that is tranferred between systems when they're at different temperatures. It is denoted as Q with SI unit Joule [J].

Heat Capacity. Heat capacity (C, [J / K]) of a substance is the amount of heat required to change its temperature by one degree. It has units of energy per degree. It is a characteristic of an object. It is defined as

$$Q = C \cdot dt \tag{D.8}$$

where Q [J] is the amount of heat supplied, C [J / K] is the heat capacity of a system or object and dt [K] is the temperature change.

Calorie (cal). Calorie is the amount of heat energy needed to raise the temperature of one gram of water by one degree Celsius at a pressure of one atmosphere.

Specific Heat Capacity. Specific heat capacity (c, [J / (gK)] is the amount of heat required to change the temperature of a mass unit of a substance by one degree. Specific heat is a more common term for the same.

The heat supplied to a mass can be expressed as

$$dQ = m \cdot c \cdot dt \tag{D.9}$$

where dQ [J, kJ] is the supplied heat, m [g, kg] is the unit of mass , c is the specific heat [J / (gK), kJ / (kgC)] and dt is the change of temperature [K]. For example: The specific heat of iron is 0.45 J / (gK), which means that it takes 0.45 Joules of heat to raise one gram of iron by one degree of Kelvin. Therefore if dQ is positive, heat is flowing into the system, and if dQ is negative, heat is flowing out of the system.

From Equation D.9 we can obtain

$$c = \frac{dQ}{m \cdot dt} \tag{D.10}$$

Specific Heat - Gases. There are two definitions of specific heat for vapors and gases:

1. $c_p = \left(\frac{\partial h}{\partial T}\right)_p$ . . . specific heat at constant **pressure**
2. $c_v = \left(\frac{\partial h}{\partial T}\right)_v$ . . . specific heat at constant **volume**

For solids and liquids, it holds that $c_p = c_v$.

Gas Constant R. The individual gas constant R can be expressed as

$$R = c_p - c_v \tag{D.11}$$

Ratio of Specific Heat The ratio of specific heat is expressed as

$$k = \frac{c_p}{c_v} \tag{D.12}$$

Latent Heat. The heat energy required to change a substance from one state to another is called latent heat. During a change of phase, the heat added does not alter the temperature. For example, if we have ice that is melting, the temperature remains at 0 °C until all the ice isn't changed to liquid water. When it finally becomes liquid water, added heat will start to increase its temperature till the next phase change (vaporization). The heat energy released when water vapor condenses to form liquid droplets is called latent heat of condensation. Conversely, the heat energy required to change liquid into vapor at the same temperature is called latent heat of vaporization [Ahr11]. Specific latent heat for a particular substance can be obtained from:

$$dQ = m \cdot L \tag{D.13}$$

where L [kJ / kg] is the specific latent heat, dQ [kJ] is the amount of energy that was released or absorbed during the change of phase and m [kg] is the mass of the substance.

E Point Sprite Rendering

In this chapter, we present the remainder of the cloud rendering algorithm. Pseudocode showing how the point sprites are drawn is presented in Algorithm 9. The most important information lies in the fact that we do not write to the depth buffer, only read from it. Furthermore, in the drawPoints function (line 11) we show that out of bounds checking (line 12) is necessary for batch sizes of variable lengths. As an example, take a situation where we have 500,000 active particles with 865 slices. This yields a batch size of $\lceil 500{,}000 \,/\, 865 \rceil \approx \lceil 578.035 \rceil = 579$. The last slice with index 864 would then have a starting particle index computed as $579 \cdot 864 = 500{,}256$ which is a particle that is already out of bounds. If the algorithm passes the first check, the second check (line 14) only recalculates the amount of particles to be drawn preventing an overflow.

Algorithm 9: Point sprite rendering

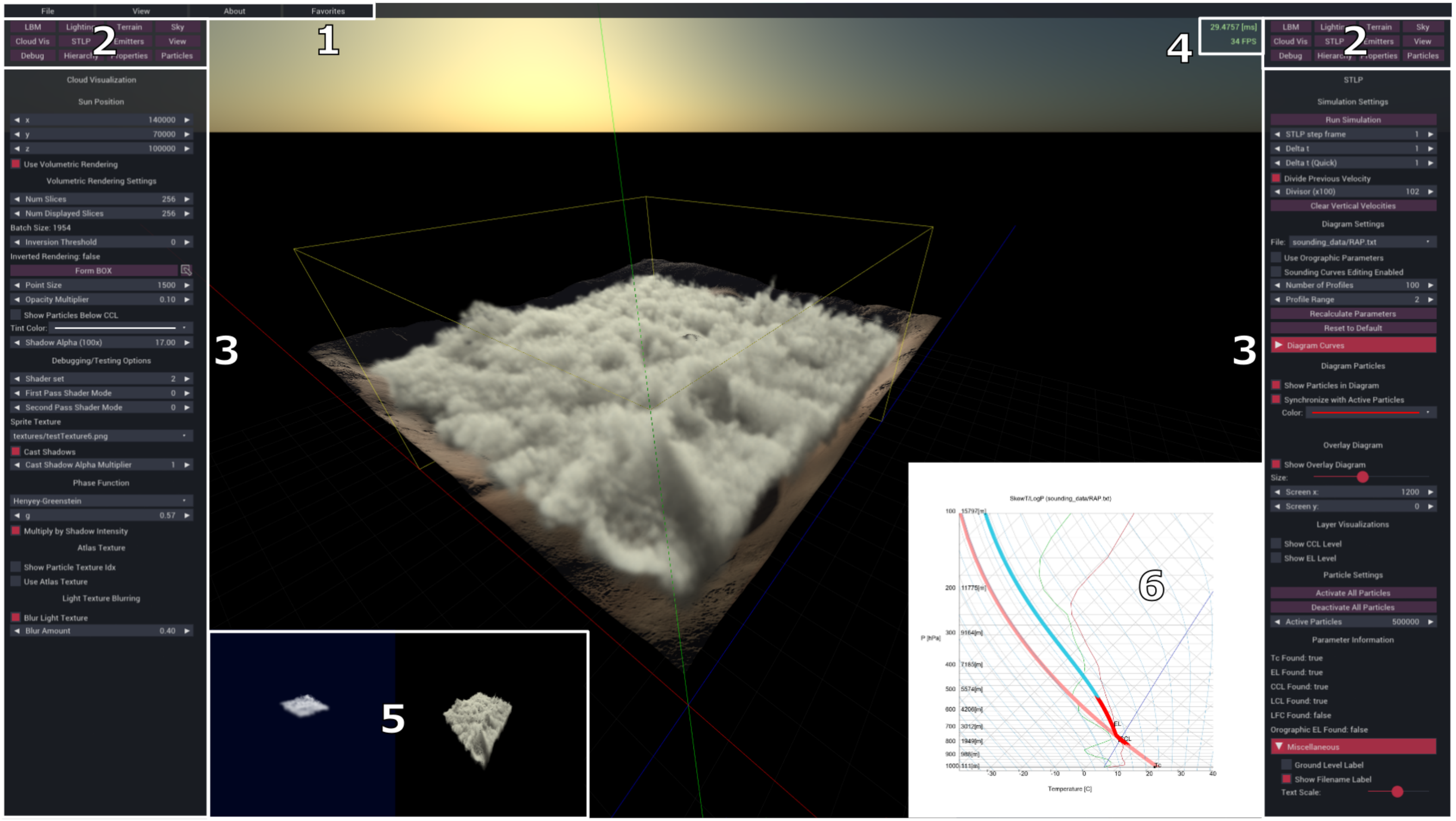
```
1  Function drawPointSprites(shader, int start, int count, bool shadowed)
2      enable depth test;
3      depth mask ← GL_FALSE;
4      enable blending;
5      use shader;
6      if shadowed then
7          bind and prepare light texture;
8      drawPoints(start, count);
9      depth mask ← GL_TRUE;
10     disable blending;
11 Function drawPoints(int start, int count)
12     if $start > numActiveParticles$ then
13         return;
14     if $start + count > numActiveParticles$ then
15         $count = numActiveParticles - start$;
16     draw points from $start$ to $start + count$;
```

F User Interface

Here we would like to familiarize the reader with the user interface of our application. The application has multiple tabs for customizing parameters of individual systems such as LBM, STLP, terrain, particle rendering and many more. There are also two viewport modes: 2D diagram viewport and a 3D viewport. Please note that both modes share the same UI panels. As shown in Figure F.1, the 3D viewport of the application offers a main menubar, two sidebars, frame rate measurement head-up display, overlay textures, and an overlay diagram.

In the 2D diagram viewport, users can navigate the STLP diagram by moving around it or by zooming in and out to view individual intersections or other details they are interested. Furthermore, curve editing (if enabled in STLP panel) is possible by dragging individual ambient temperature or dew point curve vertices along their isobars.

Each sidebar of the UI can be set to one of twelve modes that pertain to individual systems or settings. These are LBM options, lighting settings, terrain customization and settings, sky settings, cloud visualization, STLP, emitter controls, view options, debugging pane, scene hierarchy, properties tab, and particle settings. We will shortly present all these tabs in their default settings. Please note that there is a large amount of contextual and pop-up menus in the UI and not all will be covered in the next pages.


File
View
About
Favorites
1
LBM
Lighting
Terrain
Sky
Cloud Vis
STLP
Emitters
View
Debug
Hierarchy
Properties
Particles
2
3
4
5
6
29.4757 [ms]
34 FPS
SkewT/LogP (sounding_data/RAP.txt)
P [hPa]
Temperature [C]
100 15797[m]
200 11775[m]
300 9164[m]
400 7183[m]
500 5574[m]
600 4206[m]
700 3012[m]
800 1949[m]
900 986[m]
1000 111[m]
-30
-20
-10
0
10
20
30
40
EL
CCL
Tc
Form BOX
Run Simulation
Clear Vertical Velocities
Recalculate Parameters
Reset to Default
Diagram Curves
Activate All Particles
Deactivate All Particles
Miscellaneous

Cloud Visualization

Sun Position

x 140000
y 70000
z 100000

☑ Use Volumetric Rendering

Volumetric Rendering Settings

Num Slices 256
Num Displayed Slices 256
Batch Size: 1954
Inversion Threshold 0
Inverted Rendering: false
Point Size 1500
Opacity Multiplier 0.10
☐ Show Particles Below CCL
Tint Color:
Shadow Alpha (100x) 17.00

Debugging/Testing Options

Shader set 2
First Pass Shader Mode 0
Second Pass Shader Mode 0
Sprite Texture
textures/testTexture6.png
☑ Cast Shadows
Cast Shadow Alpha Multiplier 1

Phase Function

Henyey-Greenstein
g 0.57
☑ Multiply by Shadow Intensity

Atlas Texture

☐ Show Particle Texture Idx
☐ Use Atlas Texture

Light Texture Blurring

☑ Blur Light Texture
Blur Amount 0.40

STLP

Simulation Settings

STLP step frame 1
Delta t 1
Delta t (Quick) 1
☑ Divide Previous Velocity
Divisor (x100) 102

Diagram Settings

File: sounding_data/RAP.txt
☐ Use Orographic Parameters
☐ Sounding Curves Editing Enabled
Number of Profiles 100
Profile Range 2

Diagram Particles

☑ Show Particles in Diagram
☑ Synchronize with Active Particles
Color:

Overlay Diagram

☑ Show Overlay Diagram
Size:
Screen x: 1200
Screen y: 0

Layer Visualizations

☐ Show CCL Level
☐ Show EL Level

Particle Settings

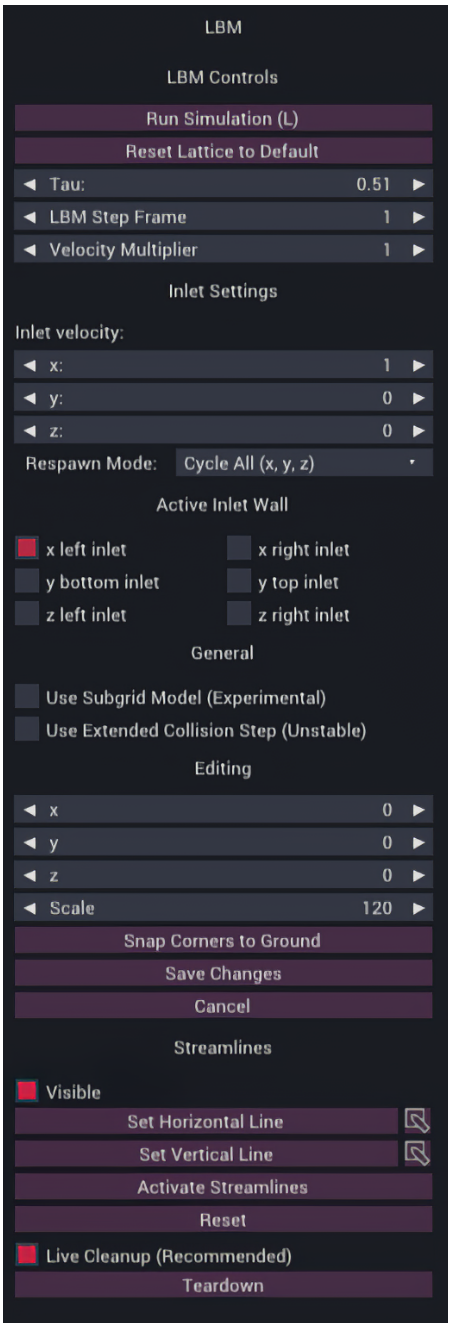
Active Particles 500000

Parameter Information

Tc Found: true
EL Found: true
CCL Found: true
LCL Found: true
LFC Found: false
Orographic EL Found: false

☐ Ground Level Label
☑ Show Filename Label
Text Scale:

Figure F.1: User interface of the application showing the horizontal toolbar (1), left and right sidebar content selectors (2), left and right sidebars (3), frame rate head-up display (4), overlay textures (5) and an overlay STLP diagram that displays particle positions on the adiabat curves as red points (6).



LBM

LBM Controls

Run Simulation (L)

Reset Lattice to Default

◄ Tau: 0.51 ►

◄ LBM Step Frame 1 ►

◄ Velocity Multiplier 1 ►

Inlet Settings

Inlet velocity:

◄ x: 1 ►

◄ y: 0 ►

◄ z: 0 ►

Respawn Mode: Cycle All (x, y, z)

Active Inlet Wall

☑ x left inlet ☐ x right inlet

☐ y bottom inlet ☐ y top inlet

☐ z left inlet ☐ z right inlet

General

☐ Use Subgrid Model (Experimental)

☐ Use Extended Collision Step (Unstable)

Editing

◄ x 0 ►

◄ y 0 ►

◄ z 0 ►

◄ Scale 120 ►

Snap Corners to Ground

Save Changes

Cancel

Streamlines

☑ Visible

Set Horizontal Line

Set Vertical Line

Activate Streamlines

Reset

☑ Live Cleanup (Recommended)

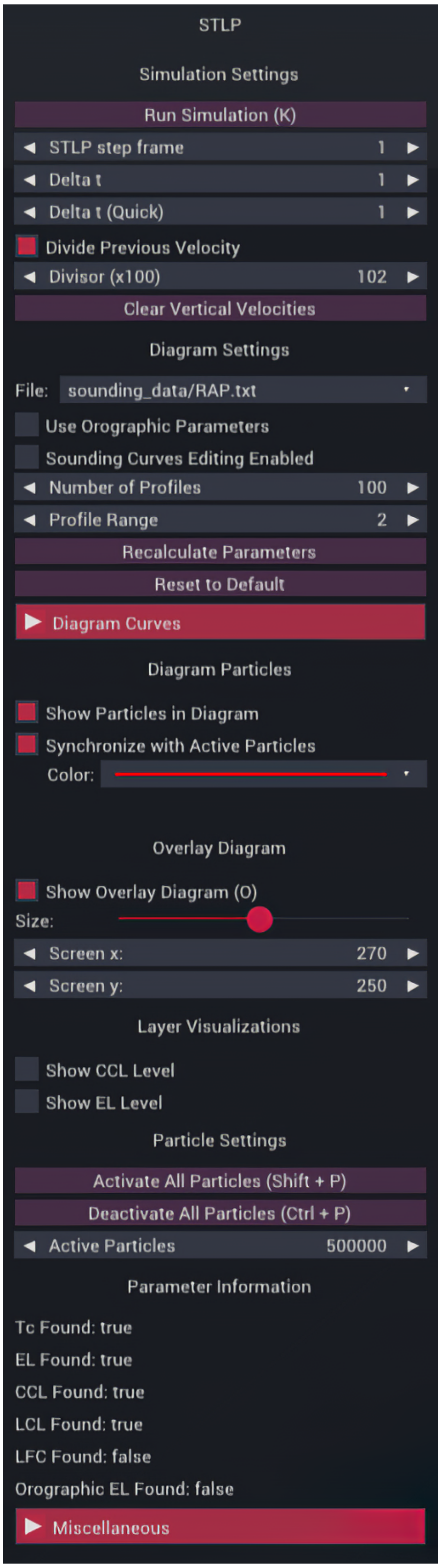
Teardown

Figure F.2: LBM Tab.

In the LBM tab, users can play or pause the simulation. Resetting the simulation is also possible by reinitializing the lattice to its default weights. Value of τ can be set. The same goes for a property named step frame. Step frame is an integer n that makes sure that the simulation is only run in frames with index that is a multiple of n. Inlet settings such as inlet velocity vector, respawn mode and active inlet walls are further customizable. Subgrid model can also be enabled and disabled at runtime. Velocity multiplier is the artificial multiplier applied when moving the cloud particles. Extended collision step as described in Equation 5.9 can also be enabled, this is however unstable in its current form.

LBM simulation area can be edited as well. The area can be positioned and scaled freely in the world space. Additionally, the area can be snapped to ground based on its four bottom corners. During editing, a secondary bounding box is shown to visualize the current simulation area. Changes made to the area can be saved or discarded as shown.

Streamlines are also managed in the LBM tab. First, streamline instance must be created by defining its streamline count and maximum streamline length. After creation, options such as setting horizontal or vertical lines of seeds are present. Additionally, tracked particles can be reset to their initial positions. Finally, the generated streamline instance can be torn down and replaced with a new one with different streamline count and length arguments.



STLP

Simulation Settings

Run Simulation (K)

◀ STLP step frame 1 ▶

◀ Delta t 1 ▶

◀ Delta t (Quick) 1 ▶

☑ Divide Previous Velocity

◀ Divisor (x100) 102 ▶

Clear Vertical Velocities

Diagram Settings

File: sounding_data/RAP.txt

☐ Use Orographic Parameters

☐ Sounding Curves Editing Enabled

◀ Number of Profiles 100 ▶

◀ Profile Range 2 ▶

Recalculate Parameters

Reset to Default

▶ Diagram Curves

Diagram Particles

☑ Show Particles in Diagram

☑ Synchronize with Active Particles

Color:

Overlay Diagram

☑ Show Overlay Diagram (O)

Size:

◀ Screen x: 270 ▶

◀ Screen y: 250 ▶

Layer Visualizations

☐ Show CCL Level

☐ Show EL Level

Particle Settings

Activate All Particles (Shift + P)

Deactivate All Particles (Ctrl + P)

◀ Active Particles 500000 ▶

Parameter Information

Tc Found: true

EL Found: true

CCL Found: true

LCL Found: true

LFC Found: false

Orographic EL Found: false

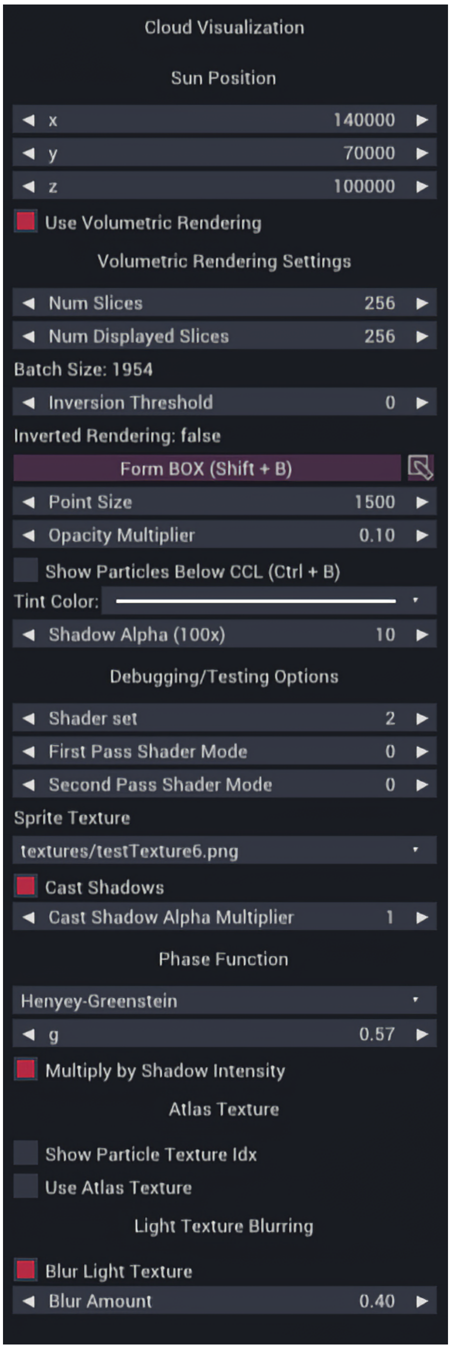
▶ Miscellaneous

Figure F.3: SkewT/LogP Tab.

In the SkewT/LogP (STLP) tab users can control both the STLP simulation as well as the diagram settings. Simulation can be played or paused. Similarly to LBM step frame, we provide an option to skip simulation step in the defined number of frames. Δt can also be set using one of the present widgets. Note that the value denoted with (Quick) only acts as a faster slider for convenience and does not have any impact on the simulation. "Divide Previous Velocity" denotes our damping factor whose 100× multiple can be set manually. The 100× multiplier is used so that users can fine-tune the simulation with high precision. An option to clear the vertical velocity (v_y) CUDA array to zeros is also provided.

In the diagram settings, users can select the sounding data file and whether to use orographic parameters. The number of convective temperature profiles to be used and their range in degree Celsius is also modifiable. Moreover, sounding curve editing can be enabled or disabled. After any changes to the diagram are made, "Recalculate Parameters" button must be used to update all curves for the simulation. If a different sounding file was selected, "Load Sounding File" button (contextual) must be pressed for the changes to take place. In the diagram curves panel, groups of curves can be hidden and their color can be changed.

Diagram particle visualization controls are also present in this tab. Diagram particles can be hidden and their active count can be synchronized with the particle system. Moreover, their visualization color can be set. Overlay diagram settings are also available. The overlay diagram can be scaled and positioned on the screen. CCL and EL levels can be visualized in the 3D viewport using their perspective checkboxes. Additionally, particle settings and parameter information panels are shown. Lastly, a miscellaneous panel gives options for scaling diagram text and changing visibility of certain diagram labels.



Cloud Visualization

Sun Position

x 140000

y 70000

z 100000

☑ Use Volumetric Rendering

Volumetric Rendering Settings

Num Slices 256

Num Displayed Slices 256

Batch Size: 1954

Inversion Threshold 0

Inverted Rendering: false

Point Size 1500

Opacity Multiplier 0.10

☐ Show Particles Below CCL (Ctrl + B)

Tint Color:

Shadow Alpha (100x) 10

Debugging/Testing Options

Shader set 2

First Pass Shader Mode 0

Second Pass Shader Mode 0

Sprite Texture

textures/testTexture6.png

☑ Cast Shadows

Cast Shadow Alpha Multiplier 1

Phase Function

Henyey-Greenstein

g 0.57

☑ Multiply by Shadow Intensity

Atlas Texture

☐ Show Particle Texture Idx

☐ Use Atlas Texture

Light Texture Blurring

☑ Blur Light Texture

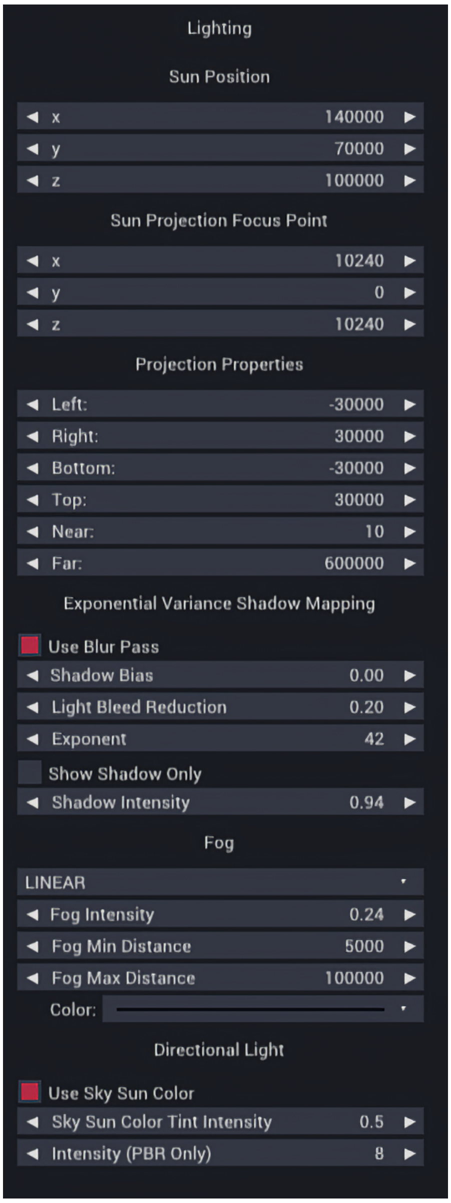
Blur Amount 0.40

Figure F.4: Cloud Visualization Tab.

In the cloud visualization tab, the sun position can be adjusted. The volumetric rendering can be enabled/disabled. In the image, options for the volumetric rendering are illustrated. In its settings, number of slices and number of displayed slices can be customized. Based on its count, the tab also shows what is the current particle batch size for a single draw call. Inversion threshold denotes a value against which the $\cos(\vec{l}, \vec{v})$ (see Algorithm 4) is compared. The panel provides information about whether the inverted rendering is currently in action. Debugging option to form a box out of the particles is also available. The box can be positioned and scaled by clicking the edit button on its right. Point size, opacity multiplier, tint color and shadow alpha determining how dark the occluded particles are, are also editable. “Shader Set” and “First/Second Pass Shader Mode” change the used shader set and debugging shader uniforms, respectively. The point sprite texture can be changed as well.

Our cloud rendering approach also supports casting shadows which can be enabled or disabled freely. Multiplier of the cast shadow can be adjusted. Phase functions are also changeable at runtime. Users can select either of the phase functions described in Section 6.6 for rendering. Appropriate settings for each phase function will be presented. An option to multiply the phase function result with shadow intensity is also available.

Experimental usage of texture atlases as sprite textures can be enabled including a visualization mode showing color coded particles based on their atlas texture indices. Lastly, light texture blurring can be enabled/disabled and its intensity can be adjusted.



Lighting

Sun Position

◀ x 140000 ▶
◀ y 70000 ▶
◀ z 100000 ▶

Sun Projection Focus Point

◀ x 10240 ▶
◀ y 0 ▶
◀ z 10240 ▶

Projection Properties

◀ Left: -30000 ▶
◀ Right: 30000 ▶
◀ Bottom: -30000 ▶
◀ Top: 30000 ▶
◀ Near: 10 ▶
◀ Far: 600000 ▶

Exponential Variance Shadow Mapping

☑ Use Blur Pass
◀ Shadow Bias 0.00 ▶
◀ Light Bleed Reduction 0.20 ▶
◀ Exponent 42 ▶
☐ Show Shadow Only
◀ Shadow Intensity 0.94 ▶

Fog

LINEAR ▾
◀ Fog Intensity 0.24 ▶
◀ Fog Min Distance 5000 ▶
◀ Fog Max Distance 100000 ▶
Color: ▾

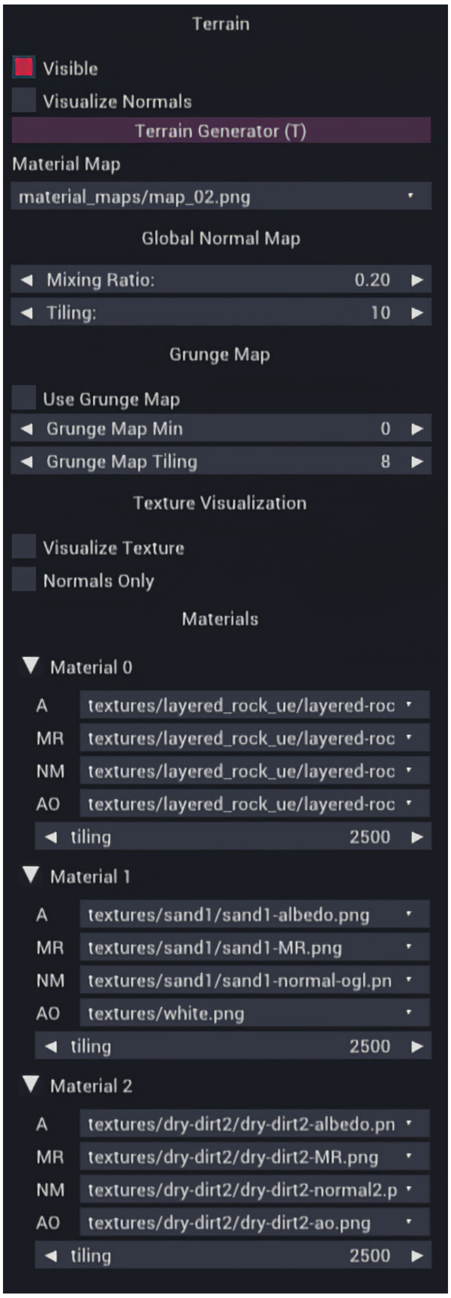
Directional Light

☑ Use Sky Sun Color
◀ Sky Sun Color Tint Intensity 0.5 ▶
◀ Intensity (PBR Only) 8 ▶

Figure F.5: Lighting Tab.

In the lighting settings tab, the sun position, focus point and the sun's orthographic projection can be customized. Furthermore, settings for the exponential variance shadow maps are present including shadow bias, whether to show shadows only, and whether to use gaussian blur on the generated shadow depth map.

Fog settings are also present. Users can select fog mode (linear or exponential) and its settings: minimum and maximum distance for linear fog, falloff for exponential fog, and its intensity and color. Lastly, directional light settings such as its intensity (only for PBR shaders) and whether to sample the sky for tinting its color are shown.



☑ Visible
☐ Visualize Normals

Material Map: material_maps/map_02.png

Mixing Ratio: 0.20
Tiling: 10

☐ Use Grunge Map
Grunge Map Min: 0
Grunge Map Tiling: 8

☐ Visualize Texture
☐ Normals Only

▼ Material 0
A: textures/layered_rock_ue/layered-roc
MR: textures/layered_rock_ue/layered-roc
NM: textures/layered_rock_ue/layered-roc
AO: textures/layered_rock_ue/layered-roc
tiling: 2500

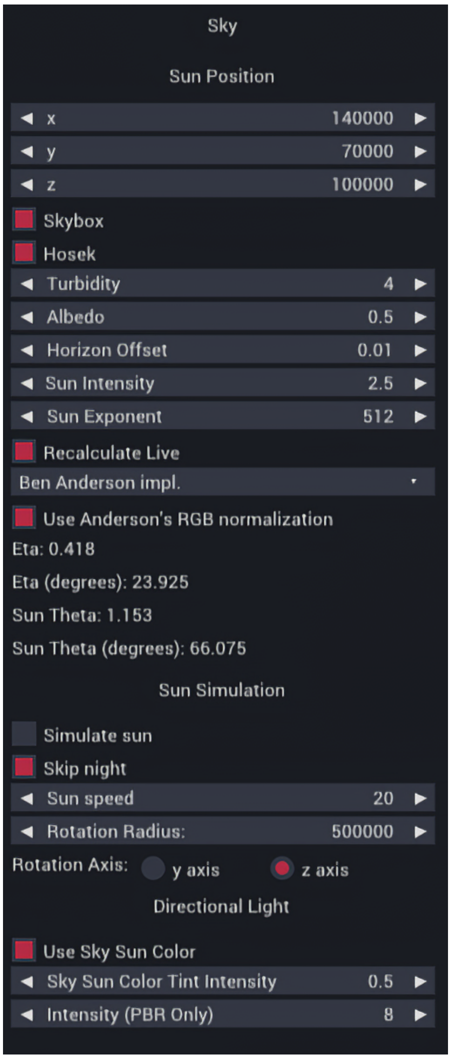
▼ Material 1
A: textures/sand1/sand1-albedo.png
MR: textures/sand1/sand1-MR.png
NM: textures/sand1/sand1-normal-ogl.pn
AO: textures/white.png
tiling: 2500

▼ Material 2
A: textures/dry-dirt2/dry-dirt2-albedo.pn
MR: textures/dry-dirt2/dry-dirt2-MR.png
NM: textures/dry-dirt2/dry-dirt2-normal2.p
AO: textures/dry-dirt2/dry-dirt2-ao.png
tiling: 2500

Figure F.6: Terrain Tab.

In the terrain tab, users can hide the terrain, choose to visualize its normals (using a geometry shader), change its material map, or set properties of its global normal map and grunge map that break repeating patterns of its tiled materials. Furthermore, users can select any texture to be projected onto the terrain for easier debugging. Normal map visualization of the terrain is also present in two modes: display default normals of the terrain mesh or display normals generated from its normal map textures. Panel of all materials is present where albedo, metallic roughness, normal map and ambient occlusion textures can be changed along with their tiling for each material individually.

Lastly, a button to open terrain generation window is present. The terrain generator window then provides options to change altitude range of the terrain and the world size of texels. Terrains can either be generated using a heightmap texture or perlin noise. If perlin noise is selected, its settings are shown such as its frequency, number of octaves, persistence (stacked multiplier applied to each octave, e.g. if set to 1/2, second octave will have intensity 1/2, third octave 1/4, n-th octave $(1/2)^n$), and its mode (basic in range [-1, 1], normalized to range [0, 1], or turbulent (absolute value)).



Sky

Sun Position

x 140000

y 70000

z 100000

☑ Skybox

☑ Hosek

Turbidity 4

Albedo 0.5

Horizon Offset 0.01

Sun Intensity 2.5

Sun Exponent 512

☑ Recalculate Live

Ben Anderson impl.

☑ Use Anderson's RGB normalization

Eta: 0.418

Eta (degrees): 23.925

Sun Theta: 1.153

Sun Theta (degrees): 66.075

Sun Simulation

☐ Simulate sun

☑ Skip night

Sun speed 20

Rotation Radius: 500000

Rotation Axis: ○ y axis ● z axis

Directional Light

☑ Use Sky Sun Color

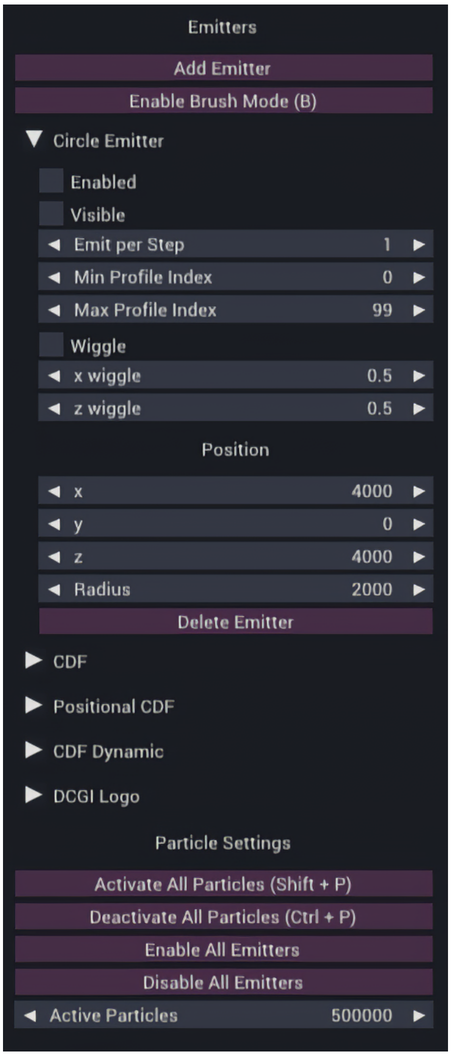
Sky Sun Color Tint Intensity 0.5

Intensity (PBR Only) 8

Figure F.7: Sky Tab.

In the sky tab, users can select whether to use a skybox instead of a single color background. When a skybox is enabled, regular HDRI images or the dynamic model by Hošek and Wilkie can be used for its rendering. If the dynamic model is selected, options such as its turbidity, albedo and sun visualization are shown. Option to recalculate the model live (each frame if its properties such as sun elevation have changed) is present and recommended to keep enabled. Debugging information such as sun elevation are shown.

Furthermore, sun movement simulation option and its parameters such as speed of movement and rotation axis (x or z) are present. Lastly, the same directional light settings as in Figure F.5 are provided.


Emitters
Add Emitter
Enable Brush Mode (B)
Circle Emitter
Position
Delete Emitter
CDF
Positional CDF
CDF Dynamic
DCGI Logo
Particle Settings
Activate All Particles (Shift + P)
Deactivate All Particles (Ctrl + P)
Enable All Emitters
Disable All Emitters

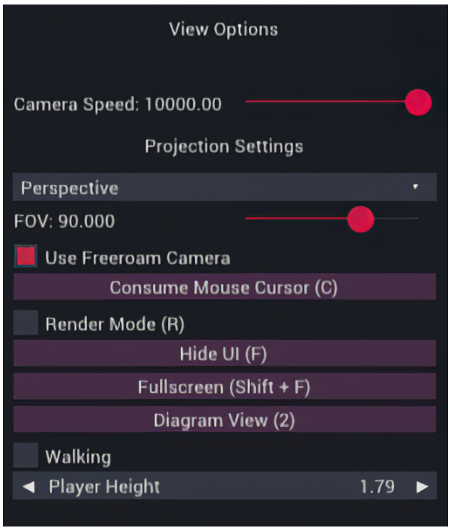
☐ Enabled
☐ Visible
Emit per Step 1
Min Profile Index 0
Max Profile Index 99
☐ Wiggle
x wiggle 0.5
z wiggle 0.5
x 4000
y 0
z 4000
Radius 2000
Active Particles 500000

Figure F.8: Emitters Tab.

In the emitters tab, users are presented with an option to create new emitters by opening an emitter creation window. Brush mode that was described in subsection 8.8.2 can also be enabled or disabled.

If the brush mode is disabled, list of all emitters is shown. Each emitter in the list can be customized individually. Customization options differ based on the emitter type. All emitters share basic options such as whether they are enabled and visible, how many particles they emit per frame and what is the range of profile indices their emitted particles fall into. In case of positional emitters, their position and scale are also modifiable.

If the brush mode is enabled, users can select the active brush from all available positional emitters. If active brush is selected, users can click and draw particles on the terrain. Using the mouse wheel, users can change the scale of the active brush. If shift is held down, the scrolling changes how many particles are emitted by the active brush. Furthermore, by holding control or the alt key, scrolling adjusts the profile indices of emitted particles.



View Options
Projection Settings
Use Freeroam Camera
Consume Mouse Cursor (C)
Render Mode (R)
Hide UI (F)
Fullscreen (Shift + F)
Diagram View (2)
Walking

Camera Speed: 10000.00

Perspective

FOV: 90.000

☒ Use Freeroam Camera

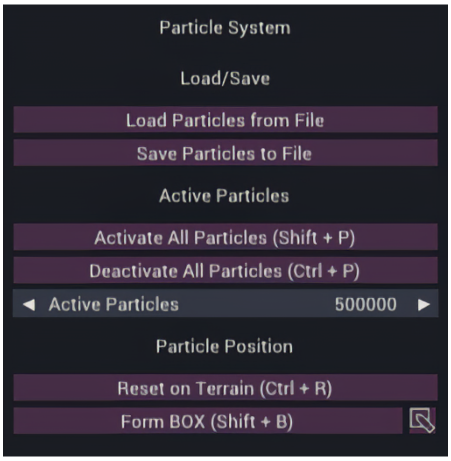
☐ Render Mode (R)

☐ Walking

◀ Player Height 1.79 ▶

Figure F.9: View Tab.

The view tab gives users basic options such as changing the camera speed, the current projection mode (perspective or orthographic), whether to use a freeroam camera, or whether the camera is snapped to ground, hence simulating walking on ground. Render mode can also be enabled which hides all helper structures such as grids or axes. Field of view (FOV) is editable when perspective projection is used. On the other hand, if orthographic projection is used, users can set front, side and top views as in popular 3D modelers like Blender or Maya. Lastly, using this panel, users can switch between the 3D and 2D viewports.

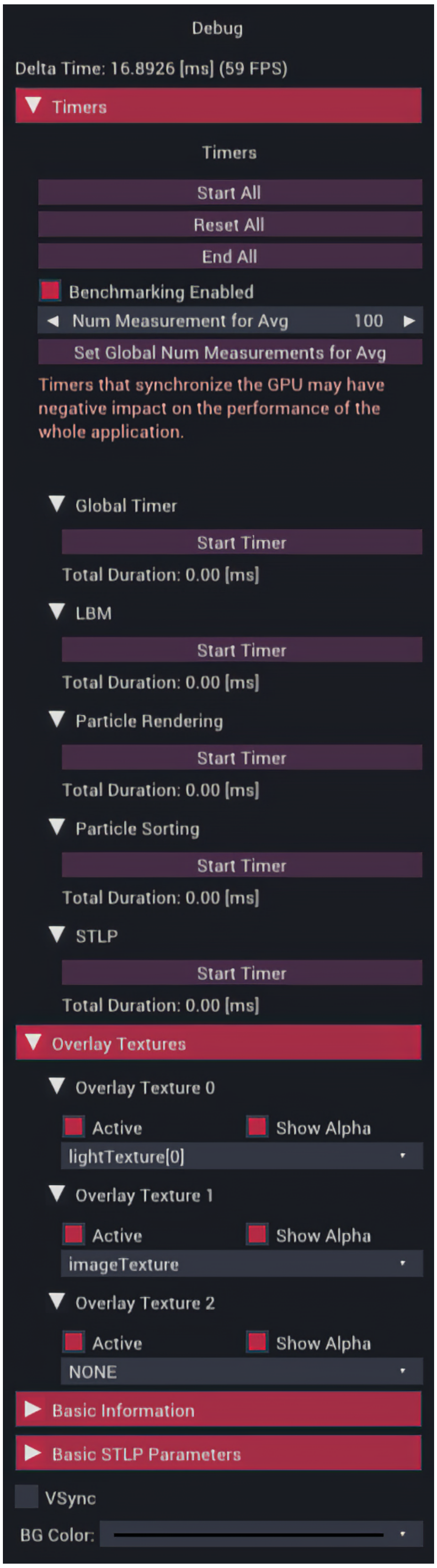


Particle System
Load/Save
Load Particles from File
Save Particles to File
Active Particles
Activate All Particles (Shift + P)
Deactivate All Particles (Ctrl + P)
Particle Position
Reset on Terrain (Ctrl + R)
Form BOX (Shift + B)

◀ Active Particles 500000 ▶

Figure F.10: Particles Tab.

In the particles tab, users can load and save current particle positions to a binary file or change the number of currently active particles. Moreover, particle positions can also be set. Particles can be either shaped into a box or respawned randomly on the ground.


Debug
▼ Timers
Timers
Start All
Reset All
End All
☐ Benchmarking Enabled
Set Global Num Measurements for Avg
▼ Global Timer
Start Timer
▼ LBM
Start Timer
▼ Particle Rendering
Start Timer
▼ Particle Sorting
Start Timer
▼ STLP
Start Timer
▼ Overlay Textures
▼ Overlay Texture 0
☐ Active ☐ Show Alpha
▼ Overlay Texture 1
☐ Active ☐ Show Alpha
▼ Overlay Texture 2
☐ Active ☐ Show Alpha
▶ Basic Information
▶ Basic STLP Parameters
☐ VSync
BG Color:

Delta Time: 16.8926 [ms] (59 FPS)

◀ Num Measurement for Avg 100 ▶

Timers that synchronize the GPU may have negative impact on the performance of the whole application.

Global Timer: Total Duration: 0.00 [ms]

LBM: Total Duration: 0.00 [ms]

Particle Rendering: Total Duration: 0.00 [ms]

Particle Sorting: Total Duration: 0.00 [ms]

STLP: Total Duration: 0.00 [ms]

Overlay Texture 0: lightTexture[0]

Overlay Texture 1: imageTexture

Overlay Texture 2: NONE

Figure F.11: Debug Tab.

The debug tab contains useful functions and information for debugging the whole application.

Its first section shows all timers for the application. Each timer can be controlled individually and it displays all its accumulated data such as average time, frame time, maximum and minimum times, and options to enable GPU synchronization before and after it starts or ends timing. The GPU synchronization is further divided into two synchronization calls: `glFinish()` for synchronizing OpenGL pipeline and `cudaDeviceSynchronize()` for synchronizing CUDA kernel execution. Timers can also be operated all at once using global start, reset and end buttons. Benchmarking checkbox can be used if the user wants the application to automatically generate `.csv` files containing all frame times captured by the timers.

Second section of the debugging tab contains controls of overlay textures. Here, users can set visibility of each overlay texture slot, select any texture to be displayed within it and whether to show its alpha texture.

Lastly, general information about the application such as terrain resolution, camera position, lattice dimensions, etc., is provided.

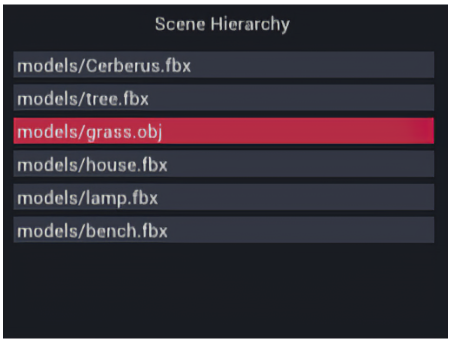
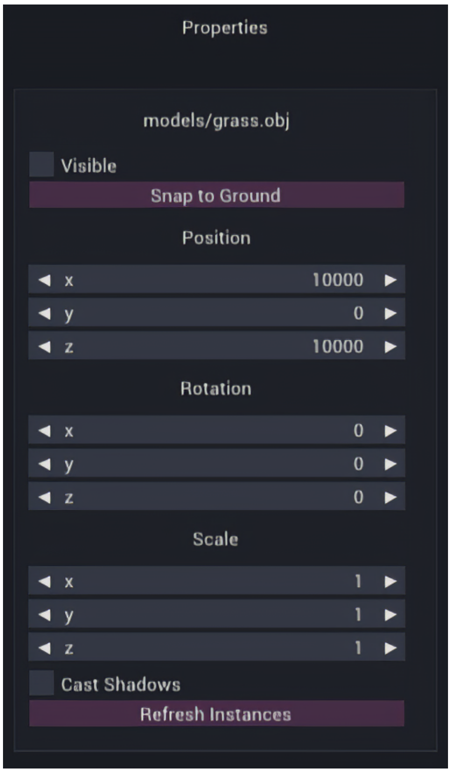

Scene Hierarchy
models/Cerberus.fbx
models/tree.fbx
models/grass.obj
models/house.fbx
models/lamp.fbx
models/bench.fbx

Figure F.12: Hierarchy Tab.

The hierarchy tab simply contains the scene hierarchy as users would expect from any other game/rendering engine. Individual objects can be selected and their detailed properties are then shown in the properties tab (see Figure F.13). Multiple objects can be selected at once and will all be shown in the properties tab.


Properties
models/grass.obj
Snap to Ground
Refresh Instances

☐ Visible

Position

◀ x 10000 ▶
◀ y 0 ▶
◀ z 10000 ▶

Rotation

◀ x 0 ▶
◀ y 0 ▶
◀ z 0 ▶

Scale

◀ x 1 ▶
◀ y 1 ▶
◀ z 1 ▶

☐ Cast Shadows

Figure F.13: Properties Tab.

Object details such as its transform (position, rotation, scale) and whether the object is visible or casts shadows are shown in the properties tab. Furthermore, options such as object snapping to ground or unparenting are present. If the selected object is an instanced model, it is also possible to refresh its instances by assigning them a new set of random positions on the terrain.

G Acronyms

AMS	American Meteorology Society
ANSI	American National Standards Institute
AO	Ambient Occlusion
BGK	Bhatnagar-Gross-Krook (Equation)
CA	Cellular Automata
CDF	Cumulative Distribution Function
CCL	Convective Condensation Level
CFD	Computational Fluid Dynamics
CML	Coupled Map Lattice
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DALR	Dry Adiabatic Lapse Rate
EBO	Element Buffer Object
EL	Equilibrium Level
EVSM	Exponential Variance Shadow Maps
FBO	Framebuffer Object
FOV	Field of View
GEP	Gene-Expression Programming
GPGPU	General-Purpose Computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HDR	High Dynamic Range
HDRI	High Dynamic Range Imaging
LBM	Lattice Boltzmann Method

LCL	Lifting Condensation Level
LFC	Level of Free Convection
MSAA	Multisample Anti-Aliasing
PBR	Physically-Based Rendering
PNG	Portable Network Graphics
SALR	Saturated Adiabatic Lapse Rate
SI	Système International (d'Unités) - International System of Units
SM	Streaming Multiprocessor
STLP	SkewT/LogP
UI	User Interface
VBO	Vertex Buffer Object
VSM	Variance Shadow Maps
WMO	World Meteorological Association

H DVD Contents

`DVD Contents`

📁 `images`	Directory containing selected images of our results.
📁 `ProjectFuji_doc`	Generated Doxygen documentation for the code.
📁 `ProjectFuji_exe`	Directory containing executables.
📁 `ProjectFuji_src`	Directory containing source code.
📄 `Cap_Thesis.pdf`	Thesis text in PDF format.
📁 `Cap_Thesis_src`	LaTeX source files for this thesis.
📄 `User_Manual.pdf`	User manual for the application.