UNED
Escuela Internacional de Doctorado
EIDUNED

TESIS DOCTORAL

2019

High-Performance Algorithms for Real-Time GPGPU Volumetric Cloud Rendering from an Enhanced Physical-Math Abstraction Approach

Carlos Jiménez de Parga Bernal – Quirós

INGENIERÍA DE SISTEMAS Y CONTROL

Dr. JOSÉ ANTONIO CERRADA SOMOLINOS
Dr. SEBASTIÁN RUBÉN GÓMEZ PALOMO

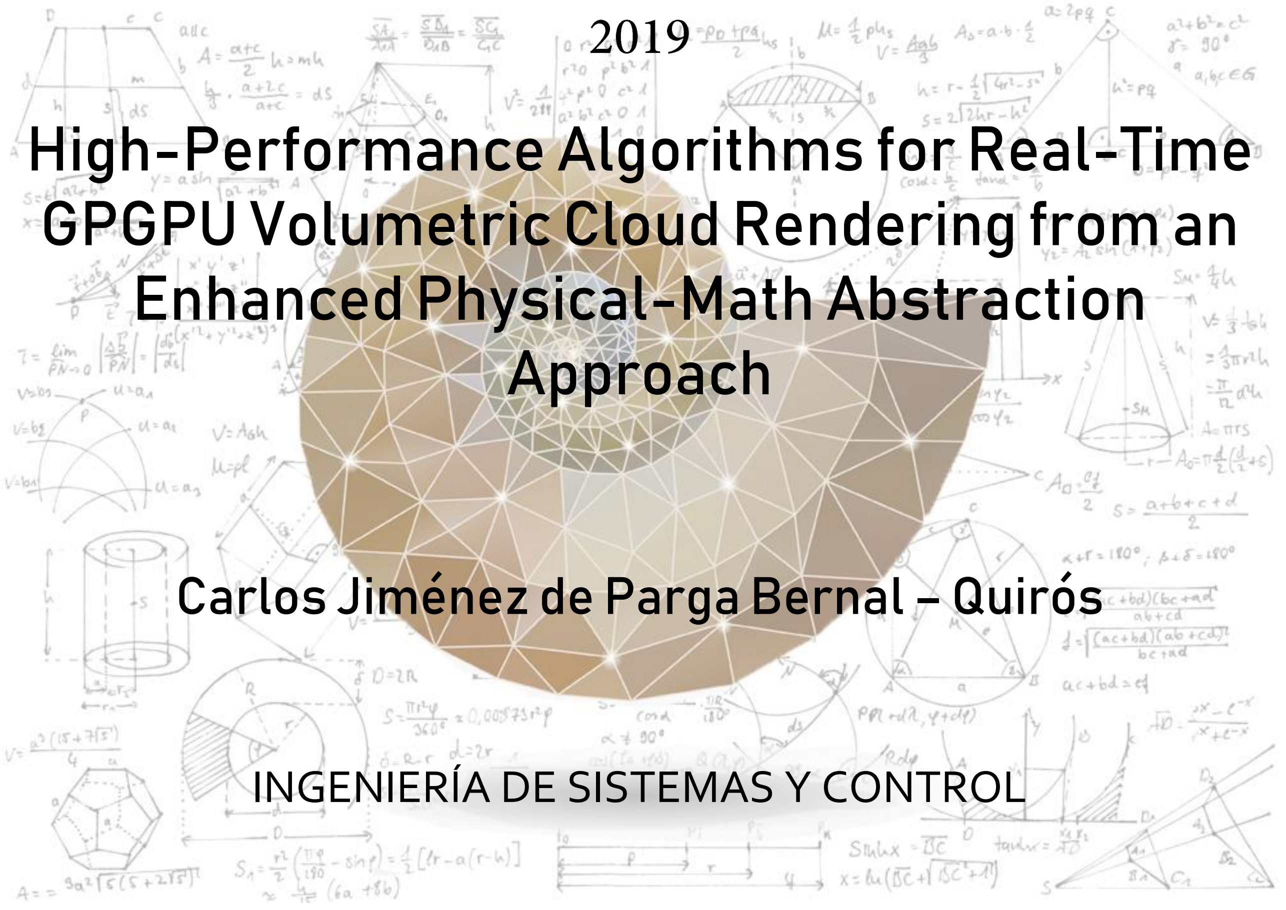
UNED

Escuela Internacional de Doctorado

EIDUNED

TESIS DOCTORAL

2019



High-Performance Algorithms for Real-Time GPGPU Volumetric Cloud Rendering from an Enhanced Physical-Math Abstraction Approach

Carlos Jiménez de Parga Bernal – Quirós

INGENIERÍA DE SISTEMAS Y CONTROL

Dr. JOSÉ ANTONIO CERRADA SOMOLINOS

Dr. SEBASTIÁN RUBÉN GÓMEZ PALOMO

Abstract

THE current trend in computer graphics applications requiring landscape rendering, such as flight simulators, computer games and educational software, is implementing realistic outdoor scenarios with a smooth interactive experience. Achieving real-time performance requires powerful hardware to render the details of the landscapes in execution time. One of the features that contributes most to realism is a cloudy sky with credible lighting and the possibility of approaching, moving around and traversing the clouds.

Since the upcoming of computer graphics in the early eighties, engineers, physicists and mathematicians like *James Blinn*, *James Kajiya* and *Geoffrey Gardner* have tried to recreate clouds analytically with both low and high rendering detail. The low-detail models were rendered in real-time in low-resolution work-stations with straightforward geometry; whereas the highly detailed ones required considerable rendering hours in those ages due to the physically oriented models with all meteorological equations. Nowadays, engineers demand the second type in real-time. Achieving this feat with those models requires massive multi-core graphics hardware that is used in the animation industry but it is expensive.

The goal of the present research is achieving a similar cloud quality with high performance using consumer-level hardware. The research follows the ontogenetic approach that performs cloud rendering using a high-level description to avoid the heavy calculations of physical-mathematical models.

In the implementation of the ontogenetic method, the first problem that this thesis resolves is the static rendering of the gaseous mass and the cloud shapes. The key contribution of this part is using modern techniques of noise sampling along with Gaussian-optimized primitives or randomized fractal clouds described in a formal language. The use of these novel techniques to reproduce the irregular nature of cumuliform clouds and the effect of light-scattering with low computing cost was crucial to achieve a quality level similar to other high-computing methods.

The second problem that this thesis deals with is the dynamic behaviour of cloud masses when they are affected by pressure and wind advection. A novel utilization of multi-core hardware for the simulation of the atmospheric fluid together with simplified algorithms of animation improve the realism in cloud deformation and translation.

In summary, this thesis demonstrates, from empiric and image quality benchmarks, that it is possible to accomplish real-time cloud rendering without expensive hardware elements and with an optimum balance between realism and performance. Besides, the actual implementation provides a reusable framework for the graphics industry.

Resumen

La tendencia actual en aplicaciones que requieren paisajes sintéticos, tales como simuladores de vuelo, videojuegos y software educativo, es implementar escenarios realistas con una fluida experiencia interactiva. La interacción en tiempo real requiere un equipamiento hardware muy potente para recrear el paisaje detallista en tiempo de ejecución. Una de las características que más contribuye al realismo es un cielo nuboso con una iluminación verosímil y con la posibilidad de acercarse, girar y atravesar las nubes.

Desde el surgimiento de los gráficos por computador a comienzos de los años ochenta, ingenieros, físicos y matemáticos como *James Blinn*, *James Kajiya* y *Geoffrey Gardner* han conseguido recrear el efecto de las nubes de forma analítica con bajo y alto detalle. Los modelos de bajo detalle funcionaban en tiempo-real en estaciones de trabajo con poca resolución utilizando geometría básica; mientras que los modelos de alto detalle requerían un elevado número de horas de cálculo en aquella época. Actualmente, los ingenieros precisan la segunda opción en tiempo-real. Para alcanzar este requisito se precisan tarjetas gráficas multinúcleo masivas que se usan en la industria de la animación pero es una solución cara.

El objetivo de esta investigación es conseguir una calidad de nubes parecida usando tarjetas gráficas al alcance de cualquier usuario. La investigación utiliza una aproximación ontogenética para la creación de las nubes usando un alto nivel de descripción para evitar los pesados cálculos de los modelos físico-matemáticos.

A partir de la citada aproximación ontogenética, el primer problema que aborda la tesis es la representación estática de las masas gaseosas y las formas de la nube. La contribución clave de esta parte es usar técnicas modernas de muestreo de ruido junto con primitivas adaptadas a las formas Gaussianas o de geometría fractal basada en lenguajes formales. La utilización de esta novedosa técnica para representar la naturaleza asimétrica de los cúmulos y el efecto de la dispersión de la luz con baja carga de cálculo ha sido crucial para alcanzar un nivel de calidad similar al de otros métodos con gran carga de computación.

El segundo problema que se aborda es el comportamiento dinámico de las masas nubosas cuando están afectadas por la presión y las corrientes de advección. La innovadora utilización del hardware gráfico para la simulación de fluidos junto con un algoritmo de animación simplificado permite mejorar el realismo de la deformación y la traslación de la nube.

Como conclusión, esta tesis demuestra, a partir de pruebas empíricas y de calidad de imagen, que es posible realizar una generación de nubes de alta calidad en tiempo-real en tarjetas gráficas convencionales con un óptimo balance entre realismo y rendimiento. Por otro lado, la implementación realizada proporciona una plataforma reutilizable para la industria de los gráficos por ordenador.

Acknowledgements

People are like clouds, turn up in our lives mottling the blue sky and then fade over the distant horizon. For this reason, I am obliged to recognise the people who helped me along my life to overcome the obstacles in the feat of my studies. Naming all of them here would be unfeasible and it is beyond my purpose.

In first place I would like to mention my parents with particular affection and remembrance for their continuous guidance, caring and support in all senses during many years of work and study. Their effort and that of my grandparents, who already enjoy a deserved eternal peace, helped me to achieve this goal. In second place I have to recognize my brother as an outstanding scientist and engineer. Without him I would not have been able to understand the mathematical background of volumetric rendering and other processes alike. It would not be fair to omit the name of some other people who helped me in particularly difficult moments during and after University, such as Gustavo Pérez, Professor Antonio Ruiz from Academia Europa, Dr. José Manuel García Carrasco and Dr. Jesús García Molina from University of Murcia. I also have a special nomination to the directors of my thesis: Professors José Antonio Cerrada and Sebastián Rubén Gómez from the Computer Systems and Software Engineering Department with the support of the doctorate programme of the Systems and Control Engineering. I would like to recognise with particular affection Professor José Antonio Ródenas for being the great Mathematics teacher who propelled me during my difficult times at school. In the words of A. Camus: in the depth of winter I found out that there was an unbeatable summer inside me.

And finally, during summer 2002 I met the person who made this feat possible. When no one gave a dime for me and I was at the bottom of the pit her regard picked me up and gave me the courage to face my fate. I finally made it to the top along with a world-class University, UNED. Thanks, Celia.

Cumulus

Still soaring, as if some celestial call
Impell'd it to yon heaven's sublimest hall;
High as the clouds, in pomp and power arrayed,
Enshrined in strength, in majesty displayed;
All the soul's secret thoughts it seems to move,
Beneath it trembles, while it frowns above.

Nimbus

Now downwards by the world's attraction driven,
That tends to earth, which had upris'n to heaven;
Threatening in the mad thunder-cloud, as when
Fierce legions clash, and vanish from the plain;
Sad destiny of the troubled world! but see,
The mist is now dispersing gloriously:
And language fails us in its vain endeavour —
The spirit mounts above, and lives forever.

Howards Ehrengedächtnis (In Honor of Howard).
Johann Wolfgang von Goethe (1749-1832).

Agradecimientos

Las personas son como las nubes, llegan a nuestras vidas, llenan el cielo azul y luego desaparecen en el lejano horizonte. Por este motivo, me parece digno reconocer a las personas que a lo largo de mi existencia y formación académica me han ayudado con los diferentes obstáculos que se me han presentado en el camino de mi estudios. Nombrar aquí a todos ellos sería inviable y nada más lejos de mi intención.

En primer lugar quisiera hacer mención con un especial recuerdo y cariño a mis padres, por su permanente compañía, afecto y apoyo en todos los sentidos durante tantos años de duro estudio y trabajo. Gracias a su sacrificio y al de mis abuelos, que ya gozan de una merecida eternidad, he llegado hasta aquí. En segundo lugar quisiera reconocer a mi hermano como un auténtico ingeniero y científico de primer nivel. Sin él no hubiera conseguido entender la base matemática de la representación volumétrica y otros tantos problemas de índole similar. No me parecería justo no nombrar algunas personas que me han ayudado especialmente en los momentos difíciles durante y después de la Universidad, tales como Gustavo Pérez y los catedráticos Antonio Ruiz de Academia Europa, Dr. José Manuel García Carrasco o Dr. Jesús García Molina de la Universidad de Murcia y por supuesto, con un gran recuerdo a mis directores de tesis: el catedrático Dr. José Antonio Cerrada y el profesor Dr. Sebastián Rubén Gómez del Departamento de Ingeniería de Software y Sistemas Informáticos y al programa de doctorado de Ingeniería de Sistemas y Control. Con especial mención y cariño al catedrático D. José Antonio Ródenas por ser un gran docente de Ciencias Exactas y por formarme en los tiempos de exclusión del colegio, porque «En las profundidades del invierno finalmente aprendí que en mi interior habitaba un verano invencible», (Camús, A.). Pues gracias a aquel verano de 2002, en el que conocí a la persona que ha hecho posible todo esto, cuando nadie apostaba nada por mí en el pozo del abismo, su mirada me hizo emerger y entablar batalla a aquella predestinación y, junto a la mejor Universidad del mundo, la UNED, he llegado por fin a lo más alto. Gracias, Celia.

Cúmulos

Todavía en alza,
como si de alguna llamada celestial
Lo impulsó a la sala más sublime del cielo;
Altas como las nubes, en pompa y poder dispuestas,
Consagrado en la fuerza, en la majestad desplegada;
Todos los pensamientos secretos del alma
[parecen moverse,
Debajo tiembla, mientras frunce el ceño.

Nimbos

Ahora hacia abajo por la atracción
[del mundo impulsado,
Eso tiende a la tierra, que había subido al cielo;
Amenazando en la loca nube de trueno, como cuando
Legiones feroces chocan y desaparecen de la llanura;
¡Triste destino del mundo agitado! pero mira,
La niebla ahora se está dispersando gloriosamente:
Y el lenguaje nos falla en su vano esfuerzo.
El espíritu monta arriba, y vive para siempre.

Howards Ehrengedächtnis (En Honor de Howard).
Johann Wolfgang von Goethe (1749-1832).

Contents

Abstract . . . i
Resumen . . . ii
Acknowledgements . . . iii
Agradecimientos . . . iv
Contents . . . v
List of Figures . . . xi
List of Tables . . . xix
List of Algorithms . . . xxi
List of Listings . . . xxiii
Acronyms . . . xxv

1 Introduction **1**
1.1 Thesis overview . . . 1
1.2 Research objectives . . . 4
1.3 Thesis synopsis . . . 5
1.4 Research contributions . . . 7
1.4.1 Main contributions . . . 7
1.4.2 Publications . . . 8
1.4.3 Nimbus framework and other software tools . . . 8
1.4.4 Conferences . . . 8
1.5 Research quality . . . 9
1.6 Summary . . . 10

2 Theoretical background **11**
2.1 Raytracing introduction . . . 12
2.1.1 Raycasting . . . 13
2.1.2 Raymarching . . . 13
2.1.3 Collision detection in raytracing . . . 14
2.2 Volumetric rendering . . . 15
2.2.1 Texture-based volume rendering . . . 15
2.2.2 Volume visualization with raycasting . . . 16

2.3 GPU rendering pipeline . . . 18
2.3.1 Fixed pipeline . . . 20
2.3.2 Programmable pipeline . . . 20
2.3.3 The graphics rendering pipeline . . . 21
2.3.3.1 Geometry stages . . . 21
2.3.3.2 Fragment stages . . . 22
2.3.3.3 External stages . . . 22
2.3.4 Use of the different shaders . . . 22
2.4 GPGPU parallel programming . . . 23
2.4.1 Introduction . . . 23
2.4.2 Parallel computing metrics . . . 24
2.4.3 Brief history of parallel computing . . . 25
2.4.4 Flynn taxonomy . . . 25
2.4.5 Single instruction, simple data (SISD) . . . 25
2.4.6 Single instruction, multiple data (SIMD) . . . 26
2.4.7 Multiple instruction, single data (MISD) . . . 26
2.4.8 Multiple instruction, multiple data (MIMD) . . . 27
2.5 CUDA architecture . . . 27
2.5.1 Entering the GPGPU . . . 28
2.5.2 CUDA in detail . . . 30
2.5.3 CUDA hardware outline . . . 30
2.6 CUDA programming model . . . 31
2.6.1 Introduction . . . 31
2.6.2 Data level parallel programming . . . 32
2.7 Cloud formation . . . 33
2.8 Cloud taxonomy . . . 34
2.8.1 High clouds . . . 34
2.8.2 Middle clouds . . . 35
2.8.3 Low clouds . . . 35
2.9 Meteorological concepts . . . 36
2.9.1 Atmospheric composition . . . 36
2.9.2 Ideal gas . . . 36
2.9.3 First law of thermodynamics . . . 37
2.9.4 Water vapour state equation . . . 38
2.9.5 Hydrostatic equilibrium . . . 38
2.9.6 Dry adiabatic lapse rate . . . 38
2.9.7 Buoyant force . . . 38
2.9.8 Stability criteria of dry air . . . 39
2.9.9 Stability criteria for moist air . . . 39

2.10 Radiometry . . . 40
2.10.1 Main radiometric cloud phenomena . . . 40
2.10.2 Phase functions . . . 42
2.10.3 Light transport model . . . 45
2.11 Fluid dynamics . . . 46
2.11.1 The Navier-Stokes equation . . . 46
2.12 Summary . . . 48

3 State of the art and author contributions 49
3.1 Ontogenetics vs. physically based methods . . . 49
3.2 Cloud rendering methods . . . 50
3.2.1 Texturized primitives . . . 50
3.2.2 Particle systems . . . 51
3.2.3 Geometry distortion . . . 52
3.2.4 Volumetric rendering . . . 53
3.3 Main thesis contributions . . . 54
3.3.1 To texturized primitives . . . 54
3.3.2 To particle systems . . . 54
3.3.3 To geometry distortion . . . 54
3.3.4 To volumetric rendering . . . 54
3.4 Pros and cons of each method . . . 55
3.5 Summary . . . 56

4 Cloud rendering 57
4.1 Water vapour emulation . . . 57
4.2 Hypertexture generation . . . 58
4.2.1 Introduction . . . 58
4.2.2 The *fBm* noise . . . 58
4.2.3 More about the *fBm* noise . . . 60
4.3 Cloud tracing . . . 61
4.3.1 Pseudospheroids . . . 61
4.3.2 Basic cloud rendering algorithm . . . 63
4.3.3 Level of detail (LOD) . . . 65
4.4 Scene delimitation . . . 66
4.5 Summary . . . 67

5 Lighting and shading 69
5.1 Introduction and contributions . . . 69
5.2 Lighting and shading model . . . 70
5.2.1 Transmittance . . . 71

5.2.2 Illumination pass (CPU side) 71
5.2.3 Render pass (GPU side) 72
5.3 The No Duplicate Tracing Algorithm (NDT) 74
5.4 Precomputed light parallelization 75
5.5 Lighting samples 77
5.6 Summary 79

6 Cloud shape improvement **81**
6.1 Gaussian cumulus cloud generation 81
6.1.1 3D Gaussian cloud generation 81
6.1.2 Case statistical analysis 84
6.2 Level of condensation 87
6.3 Fractal cumulus generation with L-systems 87
6.4 Cumuliforms with optimized metaballs 90
6.5 Cloud generation from 3D meshes 90
6.5.1 Basic operations 90
6.5.2 Ray-ellipsoid intersection approach 95
6.6 Summary 96

7 Cloud dynamics simulation **97**
7.1 Fluid simulation background 97
7.1.1 Introduction 97
7.1.2 Fluid class parallelization 100
7.1.3 Jacobi relaxation with CUDA device pointers 103
7.2 Guide points and deformation 106
7.2.1 Guide points 106
7.2.2 Deformation 108
7.3 Cloud morphing 108
7.3.1 Morphing background 109
7.3.2 Proposed model 110
7.4 Summary 114

8 Results and benchmarks **115**
8.1 Complexity analysis 115
8.2 Benchmark tests 117
8.2.1 Static rendering 117
8.2.2 Dynamic rendering and morphing 119
8.3 CUDA parallel algorithm analysis 129
8.4 Quality of rendering 130
8.5 Summary 133

9 Discussions, conclusions and future work **135**
9.1 Discussions **135**
9.2 Conclusions **137**
9.3 Future work **138**
9.4 Summary **138**

A Appendix A **139**

B Appendix B **151**

C Appendix C **166**

Bibliography and references **167**

Analytical index **175**

Copyright notice **181**

List of Figures

1.1 Example of clouds that the method is intended to develop. 1
1.2 PhD thesis schema 5

2.1 Raytracing basic layout with the main elements involved: view point (camera), Euclidean rays, light source and two solids to render. 12
2.2 A raymarching example. The lines in green are the distance estimations between the sampling points. 13
2.3 View-aligned slicing with three sampling planes. 16
2.4 A ray cast into a scalar function of a 3D volume. 17
2.5 Octree subdivision. 18
2.6 A VGA card basic block diagram (1989-2002). 19
2.7 A GPU-based graphics card block diagram (2002 to present). 19
2.8 General pipeline layout. 19
2.9 Fixed graphics pipeline (2002 and earlier). 20
2.10 Programmable graphics pipeline (2004 to present). 21
2.11 A graphical representation of the consequences of Amdahl's law. 24
2.12 (a) SISD. 26
2.13 (b) SIMD. 26
2.14 (c) MISD. 26
2.15 (d) MIMD. 27
2.16 Parallel execution of an 8-thread group. 28
2.17 Parallel execution of an 8-thread group with sample code. 28
2.18 Scalability management [nVi12]. 29
2.19 Example of thread and block distribution in two grids [nVi12]. 30
2.20 CUDA GPU hardware. 31
2.21 CUDA compilation phases. 32
2.22 CUDA data level parallel programming example. 32
2.23 Vertical profile of a cloud. 34
2.24 Flat cloud. 34
2.25 Cloud taxonomy. 35
2.26 Hydrostatic equilibrium applied to a parcel of air. 38

2.27 Scattering angle, ϕ, represented as the relation between the incident light from the sun and scattered light from the water droplet. 42
2.28 Plot of the Mie scattering angle per light intensity using 50 particles for a cumulus distribution with perpendicular and parallel polarisations. 43
2.29 Polar plot of the Mie scattering in a water droplet with 50 particles for a cumulus distribution and both polarisations. A high forward scattering appears in the image. 44
2.30 Henyey-Greenstein phase function for asymmetry parameters g = 0.99, g = 0.5, g = -0.5 and g = -0.99. 44
2.31 The effect of inscattering, outscattering, transmission and attenuation in the light transport model. 46
2.32 Horizontal forces applied on the bottom and top of the cubical droplet. 47

3.1 Virtual island [Jim03]. Example of a texturized skybox. 50
3.2 Procedural texturized skydome [Bek19]. 50
3.3 UML class diagram of a particle system [Jac08]. 51
3.4 Particle system cloud using the [Har03] method. 51
3.5 Particle system cloud using the [Hua+08] method. 51
3.6 A cumulus cloud produced by distorting a set of spheres [Gir12]. 52
3.7 A very realistic raymarched volumetric cumulus cloud [Bou08]. 53
3.8 Volumetric neural network generated cloud [Kal+17]. 53

4.1 The uniform noise in the colour scale plot shows the irregular density of water droplets in a cloud hypertexture. In the MATLAB fourth dimension plot it is possible to observe the nature of atmospheric vapour in the colour values in the cube plot. 58
4.2 Bi-dimensional representation of uniform noise (**left**) and fBm noise (**right**). It can be observed that the base noise on the left is sharper while the fBm is softer due to the octave accumulation. 60
4.3 Histogram of the uniform noise (light grey) and fBm noise (dark grey). The fBm noise has a Gaussian-like distribution. 61
4.4 Pseudospheroid shape in 2D satisfying Equation 4.9. 62
4.5 Distance/density relationship. The greater the distance between the ray and the sphere center, the lower the density of water vapour. The exponential approach gives natural realism by softening the cloud borders. 62
4.6 Three plots of different coefficients for Equations 4.10 and 4.11. The X axis represents the typical distance of cloud rendering from nearby (0) to the camera being far away (100). The Y axis represents the achieved step size at the specified distance and δ constant. 65

4.7 Bounding box delimitation of clouds. The Smits' algorithm is useful for optimizing raymarching by calculating the segment of the Euclidean straight line along which rendering must be performed according to the λ value. 66
4.8 Each cloud has a set of pseudospheres distributed in a 1D array (above). When the ray collides with a bounding box, their spheres are processed as a result of the lower and upper limit array of the pointers (below) in the GLSL fragment shader. 67

5.1 A two-pass algorithm for lighting along ray *l* and rendering along ray s 73
5.2 A basic model that illustrates the zones to sweep. In this case, only I1 to O2 and I4 to O4 are traced following the ray. 75
5.3 A flow chart illustrating the *no duplicate tracing algorithm* (lines 1–14 in Algorithm 5.2 75
5.4 The 3D grid with $M \times N \times O$ voxels with a parallel trace of the shadow including the NDT algorithm. 76
5.5 An example of shading with the sunlight coming from the left. 77
5.6 A cumulus formation over snowed mountains. 77
5.7 A sunset landscape. 78
5.8 Full moon over the ocean. 78

6.1 A cumulus plot using Equation 6.3 with $N_x(0,3)$, $N_y(0,0.9)$, $N_z(0,3)$ and $\varepsilon_2 = 2.0$. 82
6.2 Another cumulus plot using Equation 6.3 with $N_x(0,4.5)$, $N_y(0,2.9)$, $N_z(0,4)$ and $\varepsilon_2 = 2.0$. 82
6.3 A rendered Gaussian cumulus cloud with $N_x(0,3.5), N_y(0,0.9), N_z(0.3.0)$ and $\varepsilon_2 = 1.5$. 83
6.4 The generated histogram for the P_x coordinate of Table 6.2. 84
6.5 Q-Q diagram for the P_x coordinate of Table 6.2 plotted in R. 85
6.6 Box-and-whisker plot of the P_x coordinate with the other three samples. . . . 86
6.7 Flat cumulus by varying the voxel precomputed light value relative to the reduced height of the grid. 87
6.8 After each recursive call, the interpreter generates a new proportional random radius and length for the primitives. This produces more natural and impressive cumuliforms. 88
6.9 Iterations $= 7$, $\delta = 10°$. The lower the δ angle, the thinner and more strange the cloud shape becomes, so the cloud looks like a 3D spiral. This is caused by the recursive turning operators in the grammar productions that respond to the director rotation angle and the uniform random sphere radius. 89

6.10 Again, using exactly the same grammar and iterations with a yet larger angle, for instance $[50°,100°]$, the resulting grammar derivation generates dense cumulus formations. 89
6.11 Two cumulus formations by using Gaussian distribution and optimized metaballs. Just six spheres were used to render the samples by randomizing the sphere radius. 91
6.12 Ellipsoid scaling process. The proposed algorithm multiplies each triangle vertex (P_1,P_2,P_3) by a factor that typically falls in the range (0.1,2], once the barycenter (B) is calculated. Afterwards, the shader algorithm uses the maximum distance from the barycenter to the scaled triangle vertex as relation for density estimation through the ellipsoid/ray collision equations. 92
6.13 After rotation. As seen in the image above, showing the original ellipsoid in black and the resulting one in green, the previous equations allow the overlapping of the R^3 direction vectors. Hence and according to the direction of the larger triangle vertex, the algorithms produce the resulting rotation. 92
6.14 A hand mesh transformed into a soft 3D cloud. The final result is successfully optimized for the real-time GPU algorithm animation and rendering. 93
6.15 A rabbit mesh with 370 triangles. 80% decimation has been performed on this mesh, reducing the number of triangles from 1850 to 370 to achieve a suitable real-time performance. 94
6.16 The rabbit mesh blended with the ground of the landscape. The rotational filter allows to recognize the mesh correctly. 94

7.1 The velocity vector components are stored in a 3D grid of cells. 99
7.2 First step. 105
7.3 Second step. 105
7.4 Third step. 105
7.5 Fourth and final step. 105
7.6 Four cumulus clouds with guide points drifting with the generated wind inside the whole fluid grid tunnel. 106
7.7 First frame. 107
7.8 Second frame. 107
7.9 Third frame. 107
7.10 Fourth frame. 107
7.11 Fifth frame. 107
7.12 Sixth frame. 107
7.13 Seventh frame. 107
7.14 Eighth frame. 107

7.15 Since the the keyframe $i + 1$ has an additional vertex, we add one vertex c between vertex a and b in keyframe i. By using linear interpolation to generate intermediate frames, a transition from c to c' is made. 109
7.16 Case A. The excess barycenters in the hexagon are randomly distributed and overlapped over the triangle barycenters. 111
7.17 Case B. The required barycenters are added to the triangle for a random reselection to the hexagon barycenters. 111
7.18 Example layout of the rotation process for 3D meshes. 112
7.19 First step. 113
7.20 Second step. 113
7.21 Third step. 113
7.22 Fourth step. 113
7.23 Fifth step. 113
7.24 Sixth step. 113
7.25 Seventh step. 113
7.26 Eighth step. 113

8.1 With the GeForce 8800 GTS the performance at 800 $\times$ 600 pixels overcomes the limit of the hyperrealistic method shown in [Bou+08]. 118
8.2 With the GeForce 1030 GT the performance is optimum in most cases, except when the level of detail (LOD) equation is manually bypassed to force higher quality. 118
8.3 Performance in a GeForce GTX 1050 non-Ti is optimum in 99% of cases. . . . 118
8.4 Based on empirical tests in a GeForce GTX 970, the proposed model achieves a geometric frame rate increment in all algorithms. Results are very promising for the both cumulus and 3D mesh tracing algorithms in all resolutions, including Full-HD. 118
8.5 GeForce 1030 GT CPU simulation benchmark; 79.3% of the samples are above 30 FPS (see Table 8.4). The performance at 640×480 is good enough in this case. 120
8.6 GeForce 1030 GT CUDA simulation benchmark; 75% of the samples in the graph are above 30 FPS (see Table 8.5). The parallel GPU overhead behaviour is very similar to that shown in Figure 8.5. 120
8.7 GeForce 1030 GT CPU simulation benchmark with a greater problem size; 80% of the samples fall over 30 FPS as seen in Table 8.6. 120
8.8 GeForce 1030 GT CUDA simulation benchmark with a greater problem size; 80% of the samples fall over 30 FPS as seen in Table 8.7. More overall speedup is gained. 120

8.9 GeForce 1050 GTX CPU simulation benchmark; 77.7% of the samples are above 30 FPS (see Table 8.8). The performance at 800×600 is acceptable. 123
8.10 GeForce 1050 GTX CUDA simulation benchmark; 77.7% of the samples are above 30 FPS (see Table 8.9). The performance is slightly higher in this version. 123
8.11 GeForce 1050 GTX CPU simulation benchmark with a greater problem size; 72.2% of the samples are above 30 FPS (see Table 8.10). The performance is slightly lower than that of its counterpart. 123
8.12 GeForce 1050 GTX CUDA simulation measures; 75% of the samples are above 30 FPS (see Table 8.11). The CUDA version is observed to improve the CPU times. 123
8.13 GeForce 1070 GTX CPU simulation benchmarks; 100% of the samples are above 30 FPS (see Table 8.12). The overall performance is optimal. 126
8.14 GeForce 1070 GTX CUDA simulation measures; 100% of the samples are above 30 FPS (see Table 8.13). The CUDA version improves the CPU times optimally. 126
8.15 Performance of a GeForce GTX 1070 non-Ti with CPU simulations is optimal in 100% of the cases for a greater problem size as seen in Table 8.14. 126
8.16 GeForce 1070 GTX CUDA simulation metrics; 100% of the samples are above 30 FPS (see Table 8.15). The CUDA version improves the CPU times optimally. 126
8.17 Speedup calculation for the nVidia 1030 GT. The CPU fluid version improves the CUDA performance. 129
8.18 Speedup for the nVidia GTX 1050 non-Ti. The CPU version still performs better than the CUDA implementation. 129
8.19 Speedup benchmarks for the nVidia GTX 1070 non-Ti. Both the fluid and the light CUDA versions improve the CPU performance considerably. 129
8.20 Cirrus Castellanus. 130
8.21 Cirrus Uncinus. 130
8.22 Altostratus Undulatus. 131
8.23 Altocumulus Lenticularis Duplicatus. 131
8.24 Altocumulus Castellanus. 131
8.25 Cumulus Humilis. 132
8.26 Stratocumulus. 132
8.27 Cumulonimbus Calvus. 132
8.28 Cumulonimbus Incus. 133

9.1 Generated cloud using a particle system with Harris [HL01], Huang et al. methods [Hua+08]. The contour of the cloud and the overall realism lack accuracy. 136
9.2 A cloud modelling method by Montenegro et al. [Mon+17] that combines procedural and implicit models. 136

9.3 The proposed ontological volumetric cloud rendering method with lighting. The procedural noise improves cloud edges and fuzzy volume effects. 136

9.4 Photo-realistic clouds with all physical characteristics. The image above was generated with POV-Ray in 1 h, 18 min using the 100% of the CPU cores at 1024×768 resolution in pixels. The image below was generated with Lumion 7.5 taking around 1 s for the skydome texture. These are antithetical model examples that differ from the real-time system explained in this thesis. 137

C.1 Nimbus application framework class diagram. 166

List of Tables

3.1 Features per method. 55
3.2 Features per author. 55

6.1 Typical parameters for the proposed Gaussian equations. 82
6.2 The generated data for the mentioned cloud. 84
6.3 Estimator result using *fitdistr* in R with its corresponding approximation errors. 85

8.1 The table above shows the minimum distance from the cloud at which Full High-Definition (HD) 1920 × 1080 pixels rendering reaches 30 FPS (minimum real-time). This distance is suitable for scenarios where getting close to the surface of the cloud is required. 117
8.2 Minimum distance from the cloud at which full HD (1920×1080) reaches 30 FPS in an nVidia GTX 1070 non-Ti with a problem size of 10^3 for the precomputed light grid and $100 \times 20 \times 40$ for the fluid engine grid. 119
8.3 Minimum distance from the cloud at which full HD (1920×1080) reaches 30 FPS in an nVidia GTX 1070 non-Ti with a problem size of 40^3 for the precomputed light grid and $100 \times 40 \times 40$ for the fluid engine grid. 119
8.4 FPS metrics for the GeForce GT 1030 in CPU simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size. 121
8.5 FPS measures for the GeForce GT 1030 in CUDA simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size. 121
8.6 FPS measures for the GeForce GT 1030 in CPU simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size. 122
8.7 FPS metrics for the GeForce GT 1030 in CUDA simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size. 122
8.8 FPS for the GeForce GTX 1050 non-Ti in CPU simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size. 124
8.9 FPS for the GeForce GTX 1050 non-Ti in CUDA simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size. 124
8.10 FPS metrics for the GeForce GTX 1050 non-Ti in CPU simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size. 125

8.11 FPS benchmarks for the GeForce GTX 1050 non-Ti in CUDA simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size. 125
8.12 FPS for the GeForce GTX 1070 non-Ti in CPU simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size. 127
8.13 FPS benchmarks for the GeForce GTX 1070 non-Ti in CUDA simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size. 127
8.14 FPS benchmarks for the GeForce GTX 1070 non-Ti in CPU simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size. 128
8.15 FPS metrics for the GeForce GTX 1070 non-Ti in CUDA simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size. 128

9.1 Average frequency of other particle and volumetric systems compared with the proposed method. 137

List of Algorithms

2.1 Basic algorithm to implement raytracing. 15

4.1 Basic volumetric cloud rendering in GPU. 64

5.1 Lighting and shading. 70
5.2 No-Duplicate-Tracing. 74

7.1 Execution flow of the thesis fluid simulator. 98
7.2 Basic pseudocode of the linear solver with the Jacobi approximation. 103
7.3 fBm function modification for cloud deformation. 108

Listings

7.1 Original serial code version for advection. 101
7.2 Kernel call. 101
7.3 CUDA parallel kernel version for advection. 102
7.4 The linear solver kernel explained in Algorithm 7.2. 104
7.5 The trick that simulates lines 11-13 of Algorithm 7.2. 104
7.6 The client function. 105
A.1 Cumulus static and dynamic shader. 139
B.1 Mesh/Morphing static and dynamic simplified shader. 151

Acronyms

AI Artificial Intelligence

BIOS Basic Input Output System

CC-BY-SA Creative Commons Attribution Share-Alike

CEIG Congreso Español de Informática Gráfica

CPU Central Processing Unit

CT Computed Tomography

CUDA Compute Unified Device Architecture

fBm Fractal Brownian Motion

FPS Frames per Second

FX Special Effects

GLSL OpenGL Shading Language

GPL General Public License

GPGPU General-purpose Computing on Graphics Processing Units

GPU Graphics Processing Unit

LOD Level of Detail

MIMD Multiple Instruction, Multiple Data

MISD Multiple Instruction, Simple Data

MRI Magnetic Resonance Imaging

NDT No Duplicate Tracing

OO Object Oriented

PTX Parallel Thread Execution

RAM Random Access Memory

RAMDAC Random-access-memory Digital-to-analogue Converter

RGB Red, Green, Blue

SDK Software Development Kit

SIMD Single Instruction, Multiple Data

SISD Single Instruction, Simple Data

VGA Video Graphics Array

VRAM Video Random Access Memory

CHAPTER 1

Introduction

1.1 Thesis overview

Realistic and dynamic rendering of clouds, as seen in Figure 1.1, has become very valuable for applications such as computer games featuring outdoor scenarios, flight simulation systems and virtual reality environments. However, physics-based models of clouds need to solve the Navier-Stokes fluid dynamics equations and complex photorealistic radiometric function for lighting effects. This kind of model is usually employed in commercial films and are not practical for real-time software applications using graphics processing units (GPUs).

Figure 1.1. Example of clouds that the method is intended to develop.

For instance, the work of Kallwet et al. [Kal+17] at Disney Research provides a reference for highly realistic cloud rendering with a considerable amount of physics and calculations through the use of neural networks. On the other hand, particle-based systems (often used at the beginning of this millennium) are helpful in attaining accurate meteorological simulation systems, which implies many burdensome physical calculations as demonstrated in the works of Harris [Har+05] and Huang et al. [Hua+08]. However these previously cited systems lack the realism and animation effects of the research of sky domes by Mukina et al. [MB15].

As an alternative approach, this thesis proposes efficient methods for atmospheric cloud shape generation using a deformable geometry, called pseudospheroid, which produce the stochastic and irregular patterns of natural phenomena. Therefore, the utilization of a non-quantitative amount of pseudospheroids will serve as a baseline for a novel and simple method of primitive animation, resulting in effective realism.

One of the main features of this work is the possibility of approaching, manoeuvring around, and passing through gaseous cumuliform clouds by the use of volumetric rendering, whereas other implementations (see [MB15]) do not allow this characteristic; nevertheless, these kinds of models are being used in commercial products such as Lumion [1].

The methods of this thesis are performed using the raytracing technique, based on the work of Arthur Appel [App68] and Turner Whitted [Whi79] that has become useful in volumetric rendering. Raytracing requires many hours of primary and recursive calculations, which is a drawback for real-time process. For this reason, the investigation makes use of a multicore processing approach that has arisen in recent years and the possibility of optimizing the complexity of the algorithm related to lighting and dynamic rendering with GPU and central processing unit (CPU) programming techniques.

The advantage of implementing raytracing based on a GPU multicore is the fast generation of a frame in which hundreds or thousands of threads process the rays departing from each pixel of the screen 2D matrix. This method constitutes a new paradigm of computer graphics drawing that differs from the classic "putpixel". Thus, this original method is applied to generate volume rendering of clouds with sufficient realism.

In addition, both CPU and GPU multicores are used to create and implement a reductive version of a fluid mechanic system for cloud movement with very good performance and realistic results.

Though the GPU provides volumetric rendering calculations, it faces the problem of generating the smooth, fuzzy and frayed texture of clouds. Hence, the density of water vapour inside a cloud is simulated by applying a base algorithm to generate a gaseous body inside implicit surfaces.

To maximize the performance of the precomputing phase, an original algorithm is used to prevent duplicate raytracing inside overlapping volumes, which is called *no duplicate tracing*

[1] https://lumion.com/

(NDT). This approach achieves fewer loop iterations by discarding the overlapping parts of spheres and reduces the computing times by half, so it is used in the light precomputation phase (shadowing).

Regarding cloud radiometry research, the methods developed by distinguished authors are abstracted to generate a new and simplified but fast model of cloud lighting and scattering to be applied in this volumetric system.

The use of boundary representations based on Smits' algorithm [Smi02] for the raymarched volume, along with Euclidean equations for ray-spheroid collision detection, allows for minimal linear complexity when searching for primitives, therefore avoiding the use of hard-to-code space partitioning algorithms.

An improved cumulus cloud generation algorithm based on the Gaussian distribution simulates the stochastic nature of clouds with a new method for flattening the bottom side. To facilitate landscape designers' tasks, the method makes use of L-system fractal grammars, invented by *Aristid Lindenmayer* [PL96], to generate a great variety of cloud shapes according to the formal language random parameters in a creative fashion. Additionally, an innovative solution is provided to wrap mesh triangles in an smooth way by triangulating three-dimensional models, obtained from standard editors, to produce cloud shapes with recognizable features of entities.

Complexity analysis of the different algorithms is performed in the results and benchmarks sections of this thesis, which demonstrate that there are more benefits than disadvantages in comparison to other particle-based and slow, hyperrealistic implementations. As a result, the algorithms of this thesis are suitable for application in the graphics industry.

Finally, there are several commercial libraries and frameworks to include cloud rendering in custom projects. Two of the most powerful (but expensive) solutions are Silverlining [2] and TrueSky [3], which provide a complete and flexible software development kit (SDK) for OpenGL and DirectX with rain, snow and all kinds of cloud rendering. As a counterpart, my cloud framework is released as Creative Commons Attribution Share-Alike (CC-BY-SA) and General Public License (GPL) open and free software.

[2] https://sundog-soft.com/
[3] http://www.simul.co.uk/truesky/

1.2 Research objectives

The present thesis contributes to the computer graphics research field, especially in the volumetric rendering of atmospheric clouds by the use of specialized multicore graphics hardware. As a consequence, *the hypothesis of this thesis is that it is possible to generate new algorithmic models for cloud rendering with an optimum balance between realism and performance to be applied in the entry level graphics industry*. For this purpose, a higher level of abstraction of physics (radiometry and fluids) and maths (geometry, statistics, and fractals) is used to achieve higher performance in custom hardware equipment.

The following specific objectives detail the scope of the present research for the work described in this thesis:

- Analysing the state of the art of offline and online cloud rendering systems from a critical point of view.
- Studying all the maths and physics involved in natural cloud phenomena and generating models of representation.
- Increasing the level of abstraction in a spiral lifecycle until reaching an optimum balance between realism and performance.
- Providing a reliable solution to be deployed algorithmically in commercial software and hardware.
- Designing new features at the algorithmic level.
- Meeting the requirements of the static and dynamic rendering of clouds.
- Incorporating a C++ and GLSL (OpenGL Shading Language) shader application into the previous models.
- Debugging and optimizing the algorithms to speed up the application.
- Creating a user-friendly object oriented framework for advanced C++ users.
- Benchmarking the algorithms and the applications to confirm the initial hypothesis.

1.3 Thesis synopsis

This document is intended to demonstrate that the work is correct from the initial hypothesis to the satisfactory conclusions to state the thesis, since the scientific method has been applied along the research. This work is divided into four main parts as seen in Figure 1.2: The first part is an overview of basic theory and a review of the state of the art. The second part focuses on cloud rendering and radiometry implementation, and the third part provides a complete study of atmospheric fluids for the cloud animation. The final part deals with the results and benchmarks.

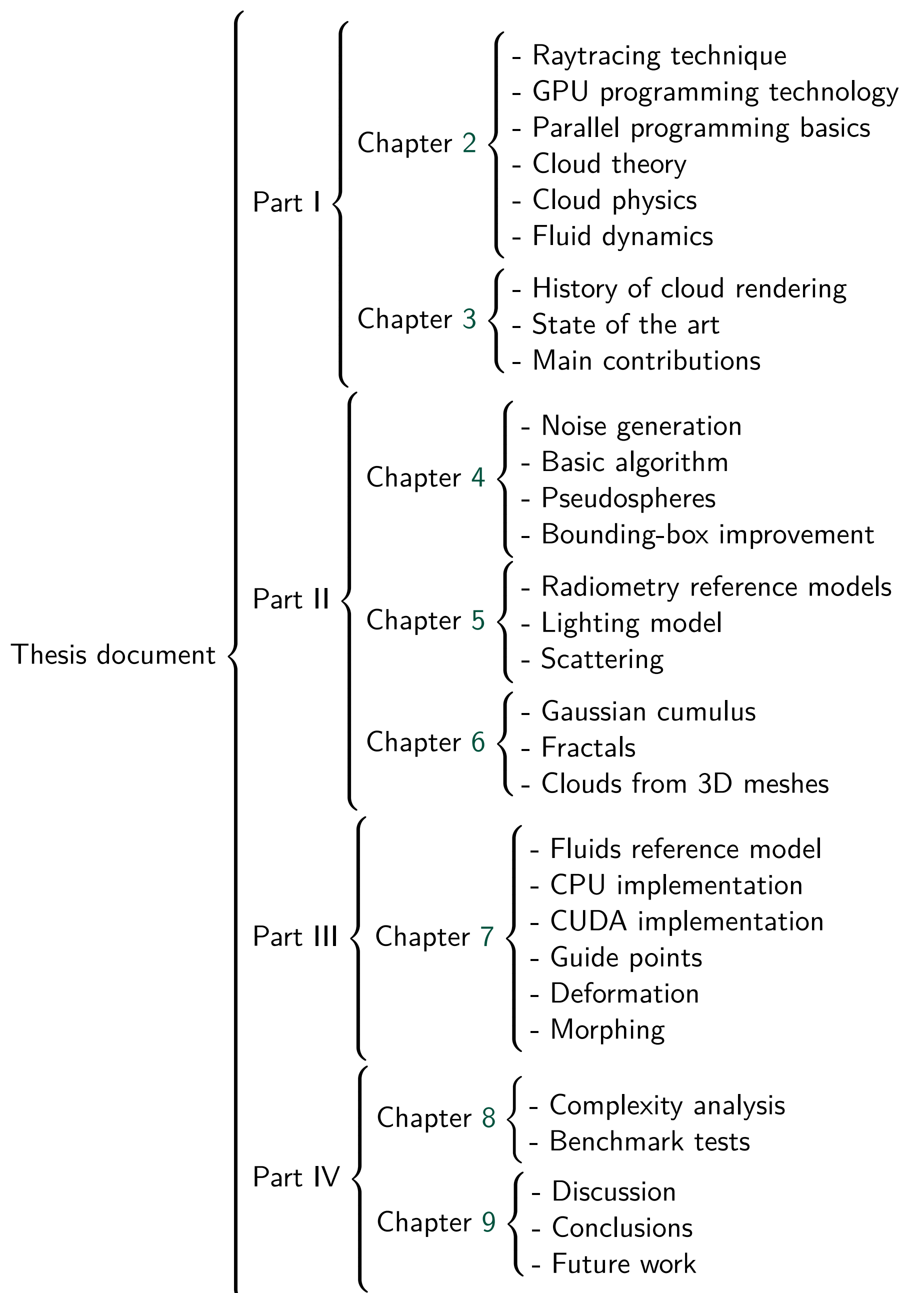


Figure 1.2. PhD thesis schema

- **PART I** (Basic theory and state-of-the-art techniques)
  - **Chapter 2:** This chapter introduces the basic theoretical background related to computer graphics techniques, volumetric rendering and GPGPU parallel programming. It also explains the basics of cloud theory and atmospheric physics.
  - **Chapter 3:** In this chapter is provided a summarized list and explanation of the main contributions of the work, the history of clouds in computer graphics and the current state of the art of cloud rendering.
- **PART II** (Cloud static rendering, texture generation and radiometry)
  - **Chapter 4:** This chapter provides the basic concepts, the approach for cloud texture generation and the basic algorithm to render clouds using a GPU. This chapter also explains the equations of pseudospheres and the improvement with Smits' algorithm.
  - **Chapter 5:** This chapter details all the concepts and contributions for cloud lighting and scattering and the work performed for speeding up implementation.
  - **Chapter 6:** The second part finalizes describing the cloud shape generation approach with the use of statistics, fractals and 3D-model-based clouds.
- **PART III** (Cloud dynamics)
  - **Chapter 7:** This part is dedicated entirely to cloud dynamics and the contribution to speed up fluid dynamics in cloud rendering with multicore GPUs by using new animation techniques.
- **PART IV** (Results, discussions, conclusions and future work)
  - **Chapter 8:** This chapter reports the results of empirical and theoretical studies to evaluate the proposed algorithms by providing benchmarks and complexity analysis of the heuristics.
  - **Chapter 9:** This chapter synthesizes all the previous content and metrics, comparing the results to those of previous work, providing objective and enlightening conclusions and proposing future lines of investigation.

1.4 Research contributions

This section exposes the main contributions to the scientific field, the software developed for other users, the published papers and other research activities.

1.4.1 Main contributions

The following are the contributions of the present work to the state-of-the-art in the generation of naturally-looking clouds. The ideas below have not been used in any prior work:

- New method to generate the geometry of the cloud based on a minimal primitive known as pseudosphere that is replicated following different algorithms depending on the desired shape.

- Lightweight pre-computing of shape and lighting in the CPU and GPU to minimize the load on processors with efficient volumetric rendering for light transmission and scattering.

- Filtering algorithms to remove unnecessary computation caused by duplicate or void ray tracing outside camera frustum, excessive level of detail and superfluous volumetric rendering that has no contribution to realism.

- Three innovative methods to generate the geometry of the cloud: Gaussian generator for cumulus, L-system generator for formal description of artistic cloud shapes and novel equations to produce clouds resembling known shapes described by 3D meshes.

- Minimal data transfer from CPU to GPU, limited to a single 64^3 single-precision floats hypertexture post-processed in the GPU to produce more realistic Fractional Brownian Motion (fBm) noise.

- A new approach of cumuliform cloud dynamics based on the GPU and CPU load distribution to solve the Navier-Stokes fluid dynamics equations with high performance and reliable results.

1.4.2 Publications

The research achievements were published in two journals as the following papers:

- *Efficient Algorithms for Real-Time GPU Volumetric Cloud Rendering with Enhanced Geometry.* Carlos Jiménez de Parga and Sebastián Rubén Gómez Palomo. *Symmetry.* 2018, Volume 10, Issue 4, Pages 1-25. JCR-Q2.
  doi:10.3390/sym10040125

- *Parallel Algorithms for Real-Time GPGPU Volumetric Cloud Dynamics and Morphing.* Carlos Jiménez de Parga and Sebastián Rubén Gómez Palomo. *Journal of Applied Computer Science & Mathematics.* Issue 1/2019, Pages 25-30, ISSN: 2066-4273, The journal is ranked as B+ by the Romanian Council for Academic Research (CNCSIS).
  doi:10.4316/JACSM.201901004

1.4.3 Nimbus framework and other software tools

All the software developed throughout the thesis is published at `http://www.isometrica.net/thesis/` in a dedicated web page that includes the following:

- The Visual C++ and GLSL implementations of the algorithms and models for real-time cloud rendering with GPU as the *Nimbus framework* within executable demonstrations.

- MATLAB code of prototypes and complexity analysis.

- Decimated 3D meshes of the Blender editor.

1.4.4 Conferences

The aim of the investigation is to translate the results and experiences to the scientific community. With this reason in mind, I attended the Congreso Español de Informática Gráfica 2019 (CEIG 2019) to present a poster about the investigations described in this thesis.

1.5 Research quality

One of the problems in checking the accuracy of the produced model is the quality. The dimensions of the quality are enumerated in the work of Sebastián, Bargueño and Novo [SBN94] as follows: features, differentiation, reliability, compliance, lifespan, technical assistance, and aesthetic. Each of these dimensions is detailed in the thesis as follows:

- **Features:** The aim is to achieve the maximum numbers of characteristics and benefits by considering efficiency, user friendliness and frames per second (FPS).
- **Differentiation:** This aspect comprises the secondary features of the product. In this case, the aim is to develop an object-oriented (OO) framework, a rotation axes, an L-system interpreter and a debug information log screen.
- **Reliability:** The code (white-box testing) is revised, and the application is debugged by searching for memory leaks and removing C++ warnings at level three. In addition, the framework is checked against different graphics hardware by performing stress tests (black-box testing).
- **Compliance:** The source code satisfies ISO C++11 compliance, and the programming guide style conforms to Java Sun/Oracle standards.
- **Lifespan:** According to thesis design precepts, the application and the framework are intended for a long lifespan.
- **Technical Assistance:** The technical assistance is managed by the author via the web page at `http://www.isometrica.net/thesis/`. This website provides email feedback and numerous details about the *Nimbus* framework to users.
- **Aesthetic:** Although this dimension is relatively subjective, it is very important in this thesis. For this reason, the aim is to achieve the maximum degree of cloud rendering realism while not overloading the system. To satisfy this dimension, morning/afternoon, sunset/dusk, starry night, and daytime landscapes are also implemented.

1.6 Summary

This chapter is a brief introduction to the forthcoming sections that will be presented in this document. Thus, in Section 1.1 a basic overview of the research is provided, presenting all the facets related to the problem of cloud rendering in offline and real-time systems and the purpose of the present investigation. In Section 1.2, the main research objectives and the hypothesis to verify them are enumerated, whereas Section 1.3 schematizes all the contents that will be explained in this thesis document. In Section 1.4, the main contributions to the computer graphics field and all the publications written during this thesis along with other activities are listed. Finally, in Section 1.5, the research quality taken into account to evaluate a work of this magnitude is discussed.

CHAPTER 2

Theoretical background

This chapter briefly explains the theoretical aspects related to the required technology for cloud rendering. Thus, in Section 2.1 the main concepts to understand the raytracing technique are introduced in addition to the two derived variants based on the reference algorithm. Section 2.2 is an overview of the basic technique used in this thesis to generate the three-dimensional volume with the application of raycasting and its optimization strategies used in this thesis.

Section 2.3 explains the basic components of a GPU graphics rendering pipeline (fixed and programmable) and the hardware background of graphics devices from the first emerging systems to the most recent ones. Section 2.4 is an overview of parallel architectures and the essential parallel programming metrics.

In Section 2.5, the Compute Unified Device Architecture (CUDA) is explained in all its essential parts. Section 2.6 is a brief reference about CUDA parallel programming used in the present work.

This chapter also reviews the main concepts and formulas about cloud and atmospheric physics. In Section 2.7, a brief introduction to cloud formation is explained, and in Section 2.8, a classification of cloud types is presented. Then, in Section 2.9, a concise explanation of the main meteorological equations of cloud generation is introduced to provide a basic understanding of the subject matter treated in this thesis. The main concepts related to radiometry and cloud lighting theory applied to cloud rendering that have been used in this thesis are clarified and introduced in Section 2.10. Finally, an overview of the Navier-Stokes and momentum equations for compressible fluid flow is introduced as a base for illustrating the cloud dynamics.

2.1 Raytracing introduction

The fundamentals of raytracing were stated by Arthur Appel [App68] and Turner Whitted [Whi79] in the last part of the twentieth century. Understanding their techniques is a starting point to formulate an efficient algorithmic model. As seen in Figure 2.1, the essential concept consists of tracing rays from each pixel in the frame buffer located at the camera position to the target objects.

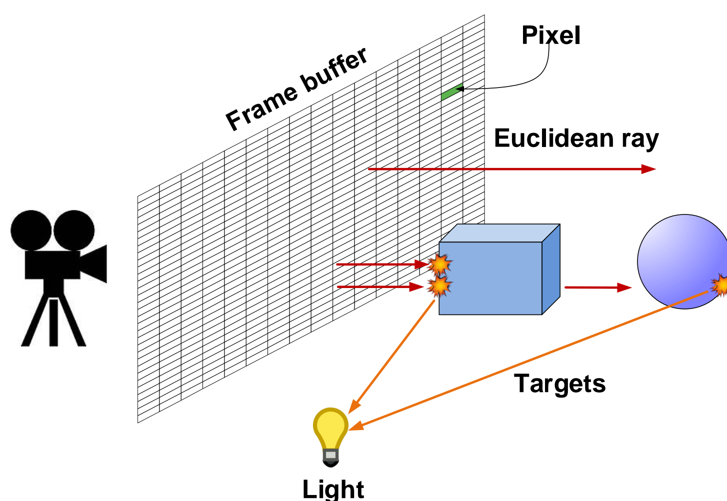


Figure 2.1. Raytracing basic layout with the main elements involved: view point (camera), Euclidean rays, light source and two solids to render.

The straight lines in red exiting from the camera are called primary rays, whereas the yellow ones are called secondary rays and are used to calculate shadows; for a mirror rendering, these rays are used to calculate reflections. The implementations of these rays are usually recursive. The raytracing results are returned in a frame buffer that is displayed in 2D on the screen.

Subsequently, the Euclidean parametric equation of a straight line [Gil88] is used in R^3 as defined in Equation 2.1 for substitution into implicit functions of basic surfaces to obtain collisions and proceed to calculate shading and lighting, according to the material sampling. Although this method is easy to understand, the current commercial implementations use accurate ray-collision methods with thousand of threads and the implementation of complex physics theory in multicore processors. The visual results produce very high realism, albeit incurring a high computational cost. Currently, raytracing can be implemented efficiently as a result of modern GPUs and the following variants: raycasting and raymarching.

$$R \equiv \begin{cases} x = x_0 + \lambda \vec{v}_x \\ y = y_0 + \lambda \vec{v}_y \\ z = z_0 + \lambda \vec{v}_z \end{cases} \tag{2.1}$$

Although raycasting, raytracing and raymarching are used interchangeably in the common technical argot, their subtle differences are detailed as follows.

2.1.1 Raycasting

In raycasting, the point of intersection with a straight line is computed analytically by using geometrical calculus. This approach is normally used along with other structures such as voxel grids and space partitioning algorithms. The method is usually applied in *direct volume rendering* for scientific and medical visualization to obtain a set of 2D slice images in magnetic resonance imaging (MRI) and computed tomography (CT).

2.1.2 Raymarching

This method is a lightweight version of raycasting in which samples are taken down a line in a discrete way to detect intersections with a 3D volume (Figure 2.2). The method is easy to implement and allows adjusting the number of samples to speed up the application; however, it lacks accurate precision rendering.

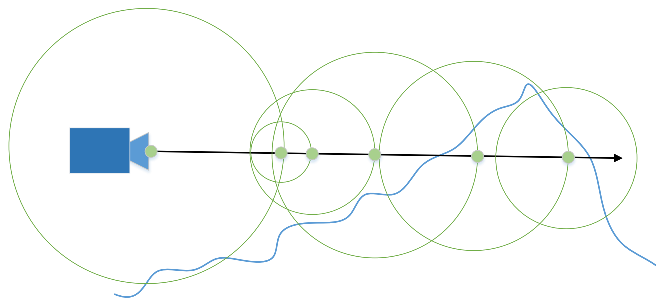


Figure 2.2. A raymarching example. The lines in green are the distance estimations between the sampling points.

In this approach is used a mixture of raycasting and raymarching for the optimization of resources as will be seen in later sections.

2.1.3 Collision detection in raytracing

The analytical part of the cloud rendering model is based in the resolution of the intersection between the ray defined in Equation 2.1 and the implicit equation of a sphere as described in Shirley's book [SK03]. Let

$$(x-c_x)^2+(y-c_y)^2+(z-c_z)^2=R^2 \tag{2.2}$$

be the implicit equation of a sphere. We can rewrite this same equation in vector form as:

$$(p-c)\cdot(p-c)=R^2 \tag{2.3}$$

If we assume the vector form of Equation 2.1 as $p(t)=o+\lambda\vec{v}$ and substitute it into Equation 2.3, we obtain:

$$(o+\lambda\vec{v}-c)\cdot(o+\lambda\vec{v}-c)=R^2 \tag{2.4}$$

Moving the terms around yields

$$(\vec{v}\cdot\vec{v})\lambda^2+2\vec{v}\cdot(o-c)\lambda+(o-c)\cdot(o-c)-R^2=0 \tag{2.5}$$

Equation 2.5 constitutes a classic quadratic equation in λ:

$$A\lambda^2+B\lambda+C=0; \tag{2.6}$$

It is known that this equation has two solutions:

$$\lambda=\frac{-B\pm\sqrt{B^2-4AC}}{2A} \tag{2.7}$$

where the discriminant is:

$$\Delta=(2\vec{v}\cdot(o-c))^2-4(\vec{v}\cdot\vec{v})((o-c)\cdot(o-c)-R^2) \tag{2.8}$$

Finally, Equation 2.8 is used to detect the collision as the next entry details:

```
1  Vector3 dir(0, 0, -1); // Ray direction vector
2  Image image(1200, 600); // Target image
   // Iterate over the pixels of the discrete surface
3  for i ← 0 to image.width do
4      for j ← 0 to image.height do
5          λ_max ← 99999
6          isCollision ← false
7          Ray ray(Vector3(i, j, 0), dir) // Ray
           // Iterate over the object list
8          for k ← 0 to objects.size() do
9              if (objects[k].detectHit(ray, .00001f, λ_max, hit)) then
10                 λ_max ← hit_λ
11                 isCollision ← true
12             end
13             if (isCollision) then
14                 image.setpixel(i, j, hit.colour); // Object colour
15             end
16             else
17                 image.setpixel(i, j, RGB(0,0,0)); // Background colour
18             end
19         end
20     end
21 end
22 image.writeImage()
```

Algorithm 2.1: Basic algorithm to implement raytracing.

2.2 Volumetric rendering

Many visual effects are volumetric in nature and are difficult to model with geometric primitives, including fluids, clouds, fire, smoke, fog and dust. Volume rendering is essential for medical and engineering applications that require visualization of three-dimensional data sets [Iki+04]. There are two methods for volumetric rendering:

- Texture-based techniques
- Raycasting-based techniques

The texture-based techniques are easily combined with polygonal algorithms, require only a few rendering passes, and offer great efficiency and quality. The second type of technique is based on simplified and optimized raytracing and is the method used in this thesis.

2.2.1 Texture-based volume rendering

Texture-based volume rendering techniques perform the sampling and compositing steps by rendering a set of 2D geometric primitives inside the volume, as shown in Figure 2.3. Each primitive is assigned texture coordinates for sampling the volume texture. The proxy geometry is rasterized and blended into the frame buffer in back-to-front or front-to-back order. In

the fragment shading stage, the interpolated texture coordinates are used for a data texture lookup step. Next, the interpolated data values act as texture coordinates for a dependent lookup into the transfer function textures. Illumination techniques may modify the resulting colour before it is sent to the compositing stage of the pipeline [Iki+04].

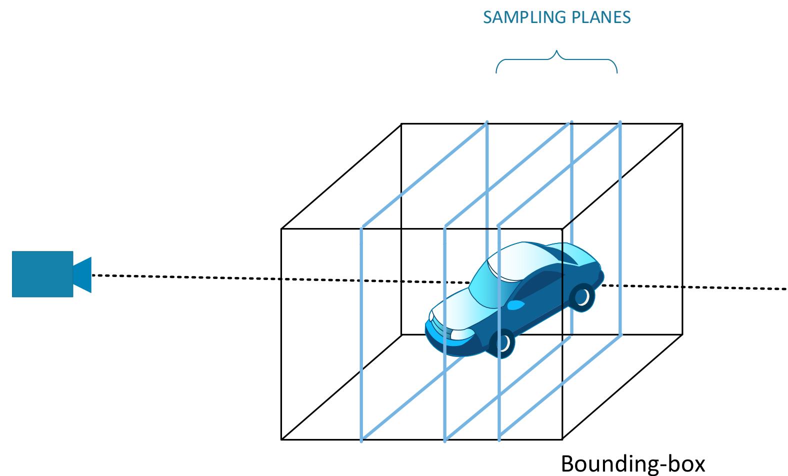


Figure 2.3. View-aligned slicing with three sampling planes.

2.2.2 Volume visualization with raycasting

As cited by Pawasauskas [Paw97], «raycasting is a method used to render high-quality images of solid objects which allows visualizing sampled functions of three dimensional spatial data. It does not attempt to impose any geometric structure on it. It solves one of the most important limitations of surface extraction techniques, namely the way in which they display a projection of a thin shell in the acquisition space. Surface extraction techniques fail to take into account that, particularly in medical imaging, data may originate from fluid and other materials which may be partially transparent and should be modelled as such. Raycasting doesn't suffer from this limitation».

Most raycasting methods are based on Blinn/Kajiya models as illustrated in Figure 2.4. Each point along the ray calculates the illumination $I(t)$ from the light source. Let P be a phase function to compute the scattered light along the ray and $D(t)$ be the local density of the volume. The illumination scattered along R from a distance t is:

$$I(t)D(t)P(cos\theta) \tag{2.9}$$

where θ is the angle between the view point and the light source.

The inclusion of the line integral from point (x,y,z) to the light source may be useful in applications where internal shadows are desired.

The attenuation due to the density function along a ray can be calculated as Equation

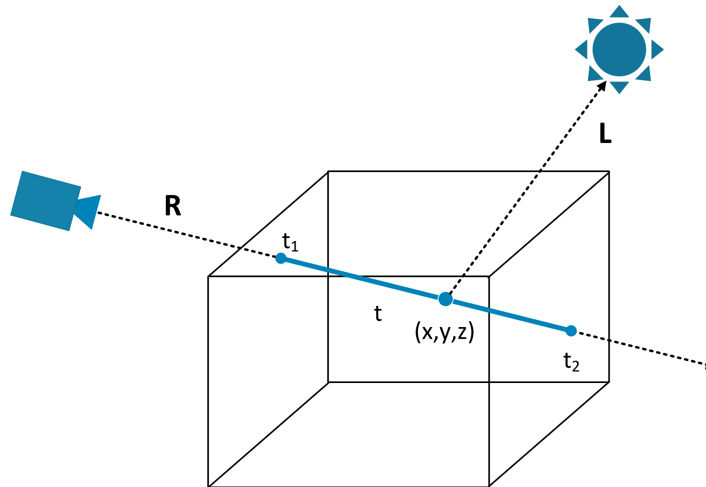


Figure 2.4. A ray cast into a scalar function of a 3D volume.

2.10:

$$e^{-\tau \int_{t_1}^{t_2} D(s)ds} \tag{2.10}$$

where τ is a constant that converts density to attenuation.

Finally, the intensity of the light arriving at the eye along direction R due to all the elements along the ray is defined in Equation 2.11:

$$B = \int_{t_1}^{t_2} e^{-\tau \int_{t_1}^{t} D(s)ds} (I(t)D(t)P(cos\theta))dt \tag{2.11}$$

Since raycasting is implemented as a raytracing variation, involving an extremely computationally intensive process, one or more of the following optimization processes are usually incorporated:

- Bounding boxes
- Hierarchical spatial enumeration
- Adaptive termination

Since bounding boxes will be explained in Chapter 4, we limit the discussion here to a brief explanation of the last two optimization methods.

In hierarchical spatial enumeration, the volume is recursively subdivided in the tree nodes in a method called Octree as seen in Figure 2.5. The identification of visible surfaces is performed by exploring the tree from front to back. Each node contains 8 child subnodes. The objective of this data structure is to search for objects in 2D/3D space in logarithmic time. The use of octrees in computer graphics, which has been applied worldwide, was invented by Donald Meagher at Rensselaer Polytechnic Institute [Mea80].

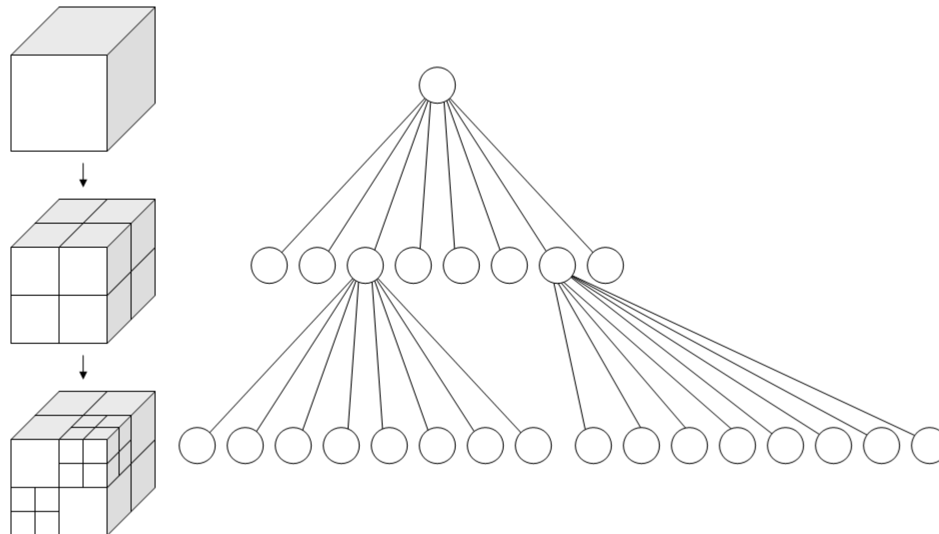


Figure 2.5. Octree subdivision.

In the adaptive termination method, the goal is to quickly identify the last sample along the ray that changes the colour of the ray significantly. This colour is usually one in which $C_{out} - C_{in} > \varepsilon$ for some small $\varepsilon > 0$. Therefore, when α exceeds 1-ε, the colour does not vary significantly. «Higher values of ε reduce rendering time, while lower values of ε reduce image artifacts producing a higher quality image» [Paw97].

2.3 GPU rendering pipeline

Since the emergence of computers, it has been useful to display data results on a visual device. The display hardware was integrated in the motherboard in the first systems and essentially consisted of a single chip that transformed data bytes to pixels on the screen. This step was performed by a random-access memory digital-to-analogue converter (RAMDAC) that converted a digital data to an electronic signal. The pioneer systems were monochromatic or green-phosphor based; later there emerged new external hardware in the form of graphic cards that connected to the motherboards via bus slots. Figure 2.6 illustrates the basic graphics card with its components: the video graphics array (VGA) graphics controller with all the digital logic to govern the data processing; the basic input output system (BIOS) with basic software for system handling; the video random access memory (VRAM) with the image data in octets; the RAMDAC and the external connectors to the screen display monitor. This first graphics devices worked only with a 2D matrix array, which received the data coming from the CPU for every frame. This simple pipeline is illustrated in Figure 2.6.

In recent years, a new type of graphics card has been developed in the industry. Recent GPUs contain advanced chips to perform fast geometry and vectorial calculations (Figure 2.7) per vertex in addition to the inclusion of a programmable pipeline, as we will see in Section 2.3.1.

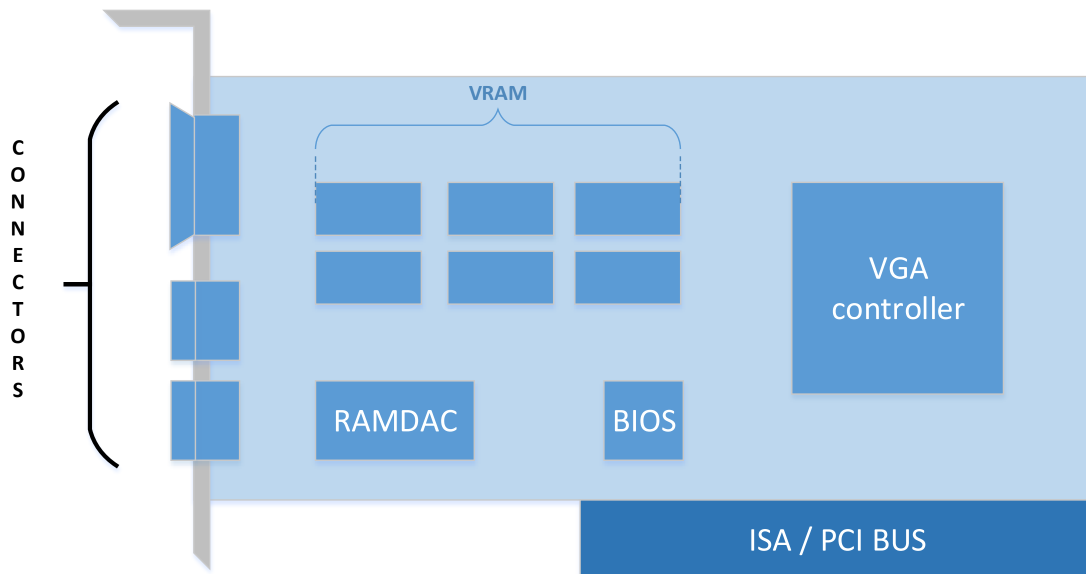


Figure 2.6. A VGA card basic block diagram (1989-2002).

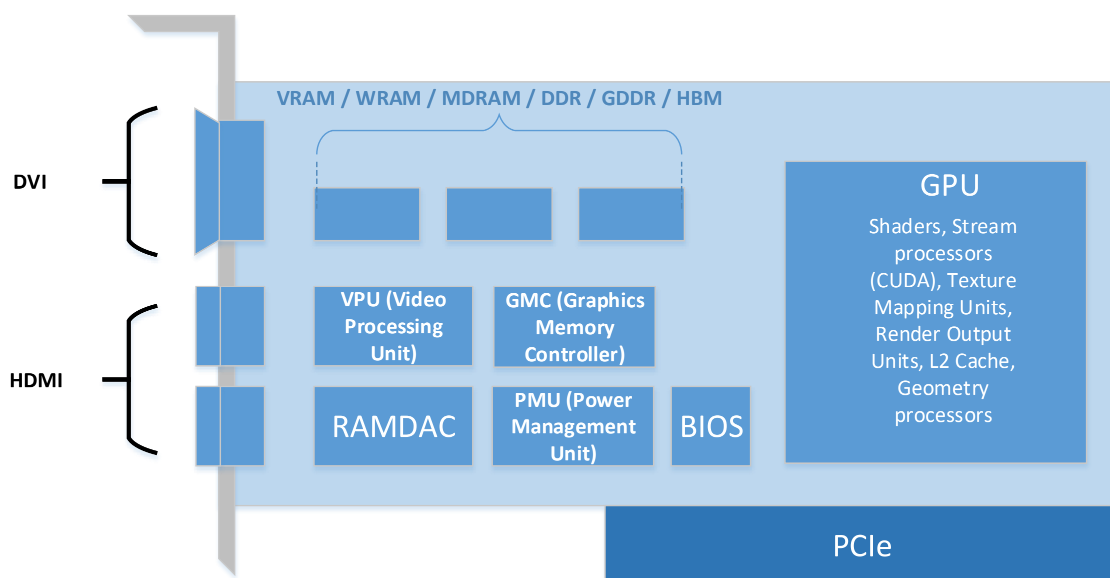


Figure 2.7. A GPU-based graphics card block diagram (2002 to present).

The basic graphics pipeline is divided in three differentiated parts as seen in Figure 2.8.

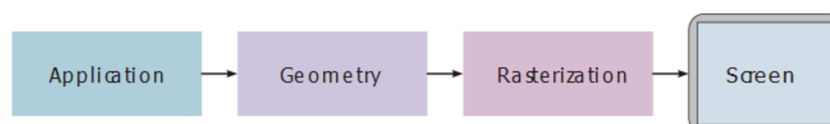


Figure 2.8. General pipeline layout.

The *application* part is associated with the graphics algorithms executed in the CPU, whereas the *geometry* part deals with vertices and transformations and, finally, the *rasterization* process fills in and filters the polygon textures. Section 2.3.3 details these parts.

2.3.1 Fixed pipeline

Essentially, the pipeline consists of a pipe into which some data are inserted into one end (vertices, textures, shaders) to travel through processors that perform very precise and concrete operations to produce the final render.

In the early OpenGL/DirectX years, the graphics pipeline was completely fixed, meaning that the data always went through the same process, in the same order in a deterministic mode.

The following is a simplified representation of the fixed pipeline, showing the data flow through the diagram block in Figure 2.9 [Rod13].

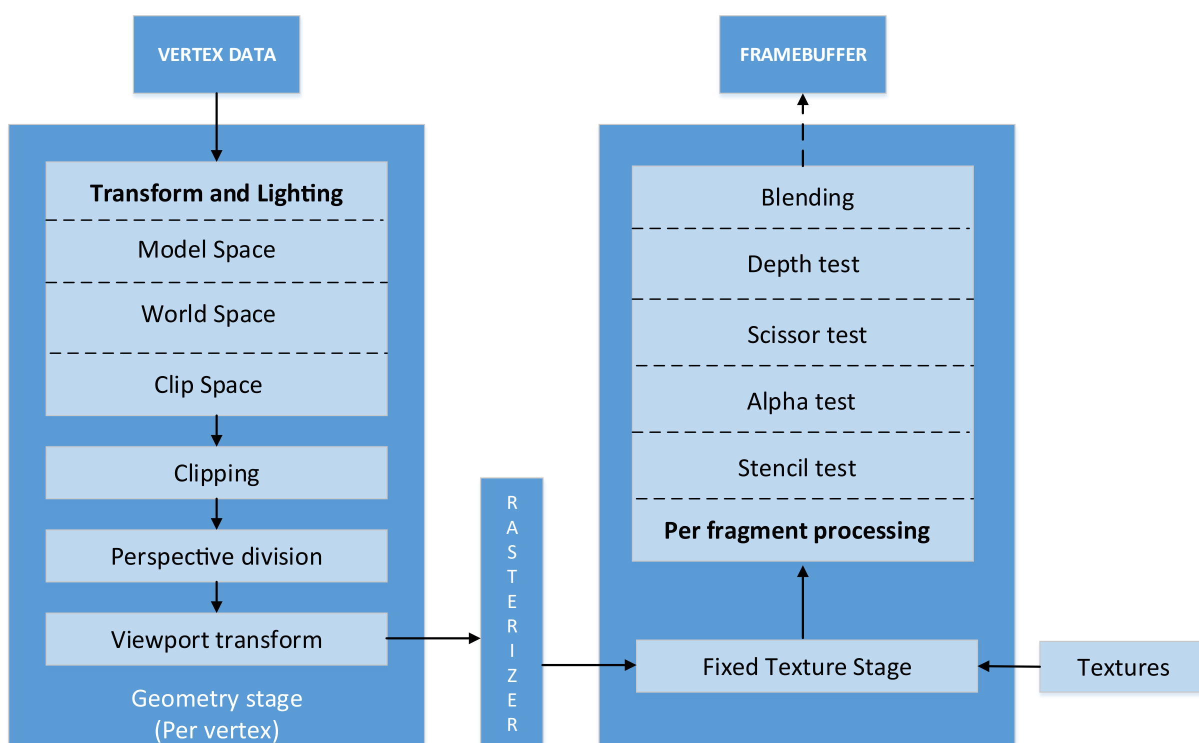


Figure 2.9. Fixed graphics pipeline (2002 and earlier).

This kind of pipeline was very popular in 2002 and earlier with the OpenGL 1.x and DirectX/Direct3D 9 previous versions.

2.3.2 Programmable pipeline

«Between the years 2002 and 2004, some kind of programmability inside the GPU was made available, replacing some of the stages seen in Figure 2.9. The first shaders programs consisted in a specific pseudoassembler language for each graphics vendor. Some of these primitive languages were Cg (from nVidia) or HLSL (from Microsoft), but they were platform specific.

During the year 2004, some companies realized the need for a high-level multi-platform

shader language. Once of the solutions was the *OpenGL Shading Language* (GLSL) which replaced two pieces of the fixed pipeline: the *vertex processing unit*, which took care of transform and lighting, and the *fragment processing unit* which was responsible for assigning colors to pixels. Those new programmable units were called *vertex shaders* and *fragment shaders* respectively» (Figure 2.10) [Rod13].

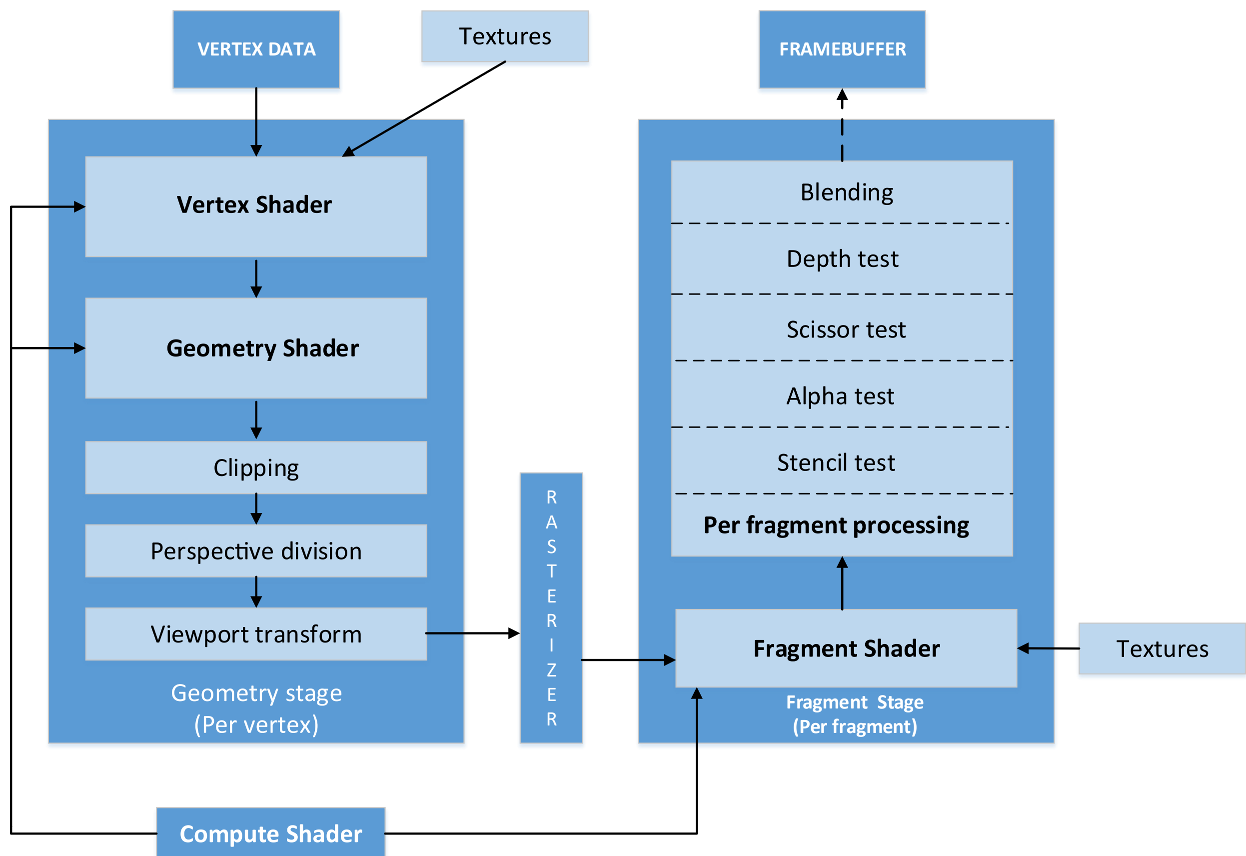


Figure 2.10. Programmable graphics pipeline (2004 to present).

2.3.3 The graphics rendering pipeline

Next, the programmable pipeline modules are explained in a summarized way.

2.3.3.1 Geometry stages

This discussion focuses on the transformation of the vertex data from the model coordinate systems to the viewport coordinate systems.

- **Vertex Data:** This is the input data of the geometry to render: vertices, normals, indices, tangents, binormals, texture coordinates, and so on.

- **Textures:** This input is usually either 2D or 3D textures to fill in or receive some other

processing step.

- **Vertex shader:** This system is responsible for the transformation of the vertices from their local coordinate system to the clip space, applying transformation matrices.

- **Geometry shader:** Using vertex shaders as inputs, this component generates new composite geometries.

- **Clipping:** This part discards the outer geometry outside the clipping space that does not fall in the view range.

- **Perspective division:** This operation converts the frustum (a truncated pyramid) into a regular and normalized cube.

- **Viewport transform:** The near plane of the normalized cube is translated and scaled to the viewport coordinates (screen).

- **Rasterizer:** This stage transforms vectorial data to a discrete representation to be processed in the next phase.

2.3.3.2 Fragment stages

This stage is responsible for rasterizing the transformed geometry for presentation on the screen.

- **Fragment shader:** In this module, textures, colours, and lights are calculated. This module is relied on intensively in this thesis.
- **Post-fragment processing:** This part calculates basic image operations such as blending, the depth test, the scissor test, and the alpha test. Finally, the results are written to the framebuffer.

2.3.3.3 External stages

Outside of the previous large blocks lies the compute shader stage. This stage can be used to control other programmable parts of the pipeline.

2.3.4 Use of the different shaders

Shaders are the programmable units used to control rendering. They act as rendering routines ready to respond to a frame iteration call. In GLSL, the shaders are written in a C-based programming language and are classified into the following types [Rod13]:

- **Vertex shaders:** With this kind of shader, we can transform the geometry by applying the model, view, and projection matrices to each vertex. In addition, the shader can be applied to evaluate noise, calculate normals, and evaluate texture coordinates. It is one of the most important shader types.

- **Fragment shaders:** This shader fills in each part of the primitive area according to a red-green-blue (RGB) colour and transparency. This shader is the most important piece in this thesis since processing a quad of 1200×600 pixels implies a total retrace and calculation of 720.000 calls to this shader. In reality, a fragment shader is executed millions of times, so the optimization process in a fragment shader is critical. Appendix A and Appendix B provide examples of GLSL fragment shaders.

- **Geometry shaders:** This shader builds new basic rendering primitives starting from the output of the vertex shaders.

- **Compute shaders:** This shader can be written to augment the vertex, fragment, or geometry shaders. It controls the programmable pipeline as seen before.

This thesis makes use of *vertex shaders* and *fragment shaders* exclusively.

2.4 GPGPU parallel programming

In addition to the programmable graphics rendering capability of a GPU, one of the most significant applications of this technology regards multicore parallel programming with the aim to solve complex problems that are difficult to accommodate with a single CPU computer. Currently, the scope of general-purpose computing on graphics processing units (GPGPU) reaches as far as the fields of big data, fluid simulation, pharmaceuticals, medicine (genome, image diagnosis), engineering, seismology, climate change, financial and economic modelling, etc.

2.4.1 Introduction

Several decades ago, software was always written in a serial manner, according to structured programming paradigms. The objective of the serial code was execution in a computer with a single CPU, where the problem was divided in discrete parts of instructions executed one by one. In contrast, in parallel programming, the problem is divided into computational parts with the aim of being executed in several CPUs. This discrete parts are solved concurrently, and each part is divided into a set of instructions ready to be executed simultaneously in different processing units. Thus, the computing time is much shorter because the problem is separated into individual parts. Each of these parts can be considered program units executing themselves in parallel.

2.4.2 Parallel computing metrics

One of the most studied parameters in parallel programming is the concept of *speedup*. If we consider t_s as the time to execute an algorithm sequentially and t_p as the time of the parallel version, the speedup is defined as the ratio $S_{up} = \frac{t_s}{t_p}$. In theory, the speedup is less or equal to p; in other cases, these constraints contradict the previous definition. Sometimes the speedup is greater than p, which is called superlinear speedup.

Another important parameter is *efficiency*, which measures the useful time of the processing time and is defined as:

$$E = \frac{S_{up}}{p} = \frac{T_s}{pT_p} \tag{2.12}$$

where p is the number of processors.

However, according to Amdahl's law, in most systems the speedup does not grow linearly but tends to saturate. As a consequence, the system efficiency decreases when the number of processors is high, as shown in Figure 2.11. Another consequence is that the speedup and efficiency increase as long as the size of the problem grows [Dor+03].

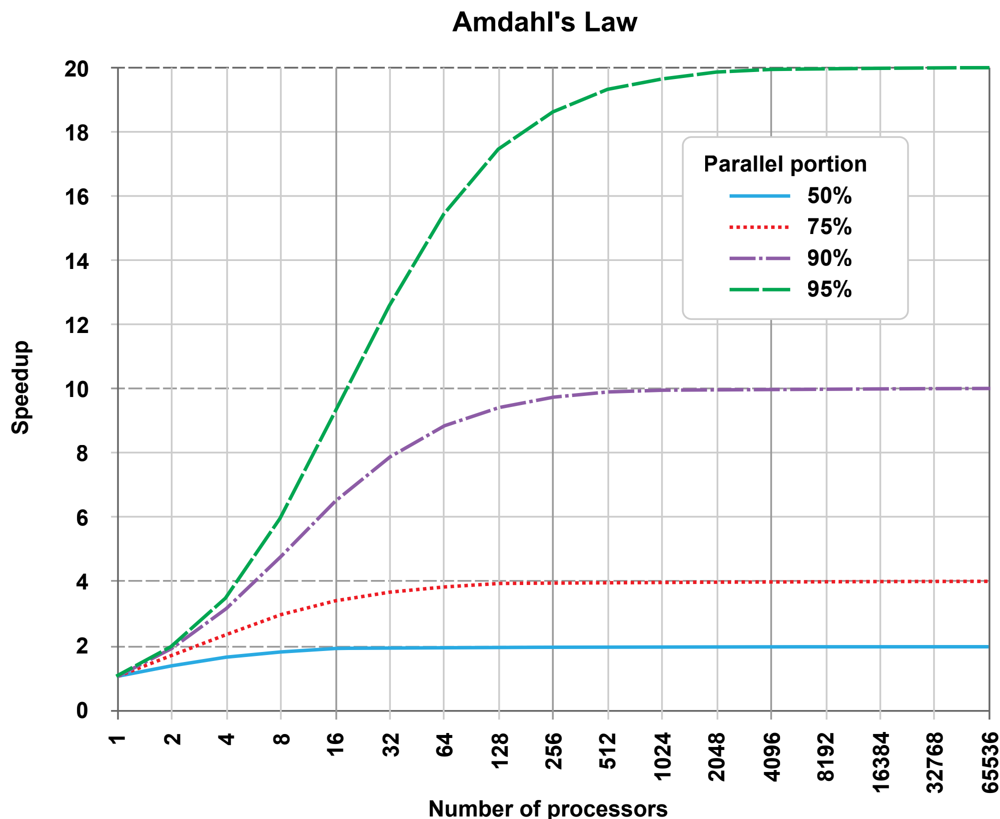


Figure 2.11. A graphical representation of the consequences of Amdahl's law.

2.4.3 Brief history of parallel computing

The history of parallel computing is interleaved with general computing history. It is certain that the origin of computing dates from the use of abacus 5000 years ago. Parallel computing in particular may be established in the middle of the twentieth century with the development of the first second-generation transistor-based computer.

In 1958, Gill stated the bases of parallel programming, and one year later, Holland proposed the possibility of executing a set of programs simultaneously. In 1963, Conway described a parallel computer design and its programming.

It was in 1981 that the first commercial parallel system appeared, the *Butterfly*, distributed by *BBN Computers Advanced*. This system was capable of dividing the workload across 256 Motorola 68000 processors interconnected through a multistage network of 500 kB of memory. Thirty-five machines were sold, mostly to universities and research centres [Dor+03].

In the 1980s and early 1990s the golden age of parallel programming began, specifically data level parallel programming. Among the most prominent architectures the *Connection Machine*, *MasPar* and *Cray* can be listed, as well as very expensive and powerful mainframes. This environment led to a dark age of parallel computing since this powerful equipment was very difficult to sell. The solution was to replace these expensive machines with massive parallel clusters and grids.

In recent years, the performance of parallel computing has significantly increased according to Moore's law. The most recent worldwide phenomenon is the democratization of parallel programming as a result of multicore GPUs.

2.4.4 Flynn taxonomy

There are several types of parallel architecture classification models; one of the most used was defined by the engineer Michael J. Flynn in 1966 and it is known as the Flynn taxonomy.

This classification distinguishes multiprocessor architectures according to two different dimensions: data and instructions, where each dimension has only two possible states: single and multiple.

Here, each of these architectures is briefly explained based as discussed by Dormido, S. et al.[Dor+03].

2.4.5 Single instruction, simple data (SISD)

In this classification falls most sequential computers used nowadays such as PCs and workstations. Instructions are executed sequentially, but they can rely on segmentation and more than one calculation unit. This approach makes use of only one control unit that governs all the instructions (Figure 2.12).

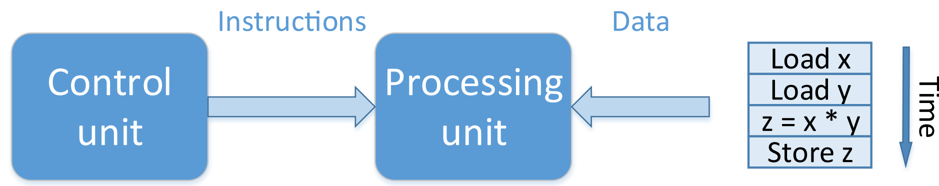


Figure 2.12. (a) SISD.

2.4.6 Single instruction, multiple data (SIMD)

In this kind of parallel architecture, all units execute the same instruction per clock cycle, and each one operates over a different part of the data. This architecture is suitable for image and graphics processing. Included in this classification are matrix processors in which more than one processing unit works over different data flows in vector form (Figure 2.13).

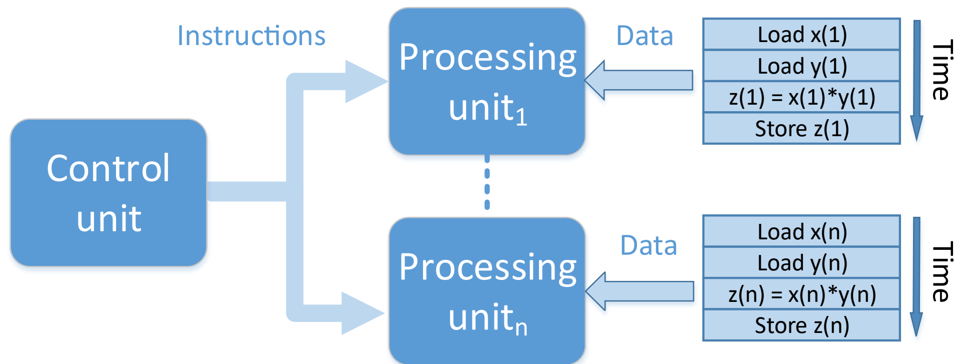


Figure 2.13. (b) SIMD.

2.4.7 Multiple instruction, single data (MISD)

In this architecture, several processing units operate over the same data flow (Figure 2.14).

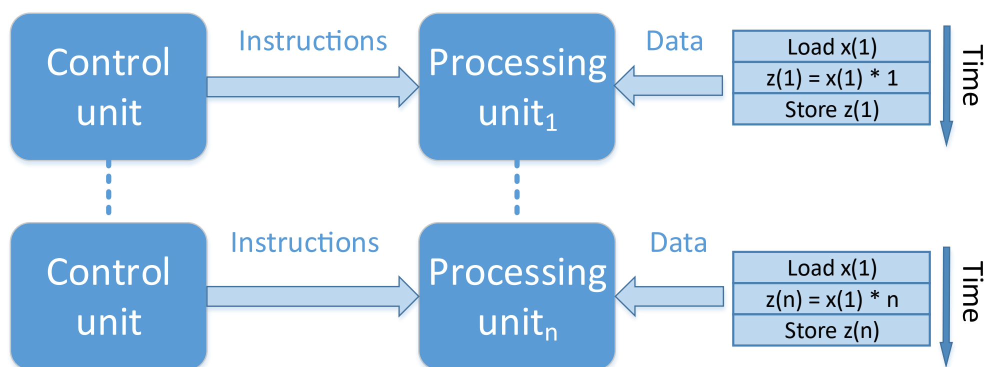


Figure 2.14. (c) MISD.

2.4.8 Multiple instruction, multiple data (MIMD)

Into this category fall most multiprocessor and multicomputer systems, in which each program is executed in a different processing unit. This state implies an interprocessor coordination since all the data flows are obtained from a shared memory space. Each processor works independently. The great advantage of SISD versus MIMD is that SISD requires less hardware since it needs only one control unit. In an MIMD model, each unit executes a copy of the program in an operating system. One advantage of MIMD is that it is compatible with conventional PCs (Figure 2.15).

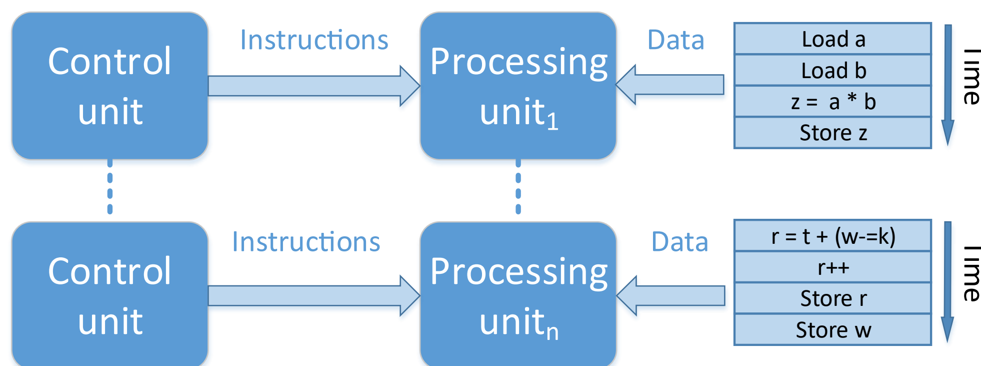


Figure 2.15. (d) MIMD.

The specific multiprocessor is built with a shared memory system and has access to the same physical directions. Each memory modification is visible to other processors through a interconnection network in a bus configuration. On the other hand, multicomputers use distributed memory configuration and communicate each other via messages passing through a fast Ethernet LAN network with switches. This system is typically implemented in heterogeneous PC or workstation networks called clusters.

2.5 CUDA architecture

In November 2006, nVidia introduces the Compute Unified Device Architecture (CUDA) with the presentation of the GeForce 8800 model. Since then, many models have arisen in the market, such as Tesla and Quadro. These models are very similar but contain different bus bandwidths, memory, cores and registers.

The new nVidia GPUs were built from a basic constructive block replication. The main advantage of the new multicore GPUs is based on a hardware mapped thread hierarchy with an efficient and transparent parallelism. Beside this aspect, this framework allows the creation, planning and transparent execution of thousands of threads concurrently.

Once of the advantages of the multicore GPUs is the reduction of workload in the CPU since this workload is sent to the previous one. Currently, the use of CUDA is in great demand in computer graphics and videogames and—more broadly—has been applied in the scientific

and engineering areas as a form of general purpose processing.

2.5.1 Entering the GPGPU

The current nVidia GPUs are characterized in their massive multicore-multithread architecture. CUDA scientists and engineers have been able to increase the calculation power by a factor of 100 in recent years.

Once of the main features of CUDA programming is the combination of host serial code implementation along with device[1] parallel code in the same program.

The execution on the device side is performed on a finite parallel thread set that acts over a different part of the data (see Section 2.4.6). Thus, the *kernel* concept can be defined as a set of multiple concurrent threads, where only one kernel is executed at a time in the device and many threads cooperate in the kernel execution. Each thread has its own identifier to ensure transparent parallelism over the data. There are two types of threads in CUDA:

- *Physical threads:* The nVidia GPU threads, where the thread creation and context interchange are essentially free.
- *Virtual threads:* A GPU core executing multiple CUDA threads.

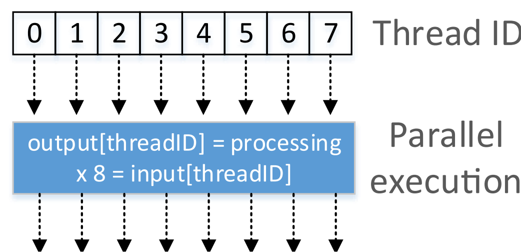


Figure 2.16. Parallel execution of an 8-thread group.

As seen in Figure 2.16, the eight threads perform data processing over the respective array zone according their thread identifier.

```
output[threadID] = input[threadID];      (1)
(2) syncthreads();
float xscale = output[threadID]*α        (3)
```

Figure 2.17. Parallel execution of an 8-thread group with sample code.

The threads involved in part (1) of Figure 2.17 must await the other threads in the group after their processing due to the *syncthreads()* primitive, part (2). Once they have been synchronized, they operate over the output, part (3).

[1]From now on, we will refer to the CPU system as the host and the nVidia GPU as the device.

Another useful term in CUDA is the concept of the *block* (Figure 2.18). Each block is a group of n threads that perform a portion of the problem task. It must be considered that the inner threads of a block can be synchronized between them, whereas the threads of the outer blocks cannot [2]. Therefore, it is considered that an implicit synchronization barrier exists among two starting consecutive kernels. As a consequence, this independence provides great scalability, where *grids* are scaled according the number of parallel cores (Figure 2.19).

Regarding the memory hierarchy, there are three types of memories in CUDA. The first one is the *local memory* dedicated for each thread. This kind of memory is thread specific and lies within the scope of a kernel function. It is the memory for the local data and is exclusively used for operations implemented in the cited function. Second, it is the *shared memory* for the threads. This memory type allows per-thread data interchange when they are executing in the same kernel, enabling performance optimization. Finally, the *global memory* is room reserved for input data storage and kernel results as explained by Sanders and Kandrot [SK10].

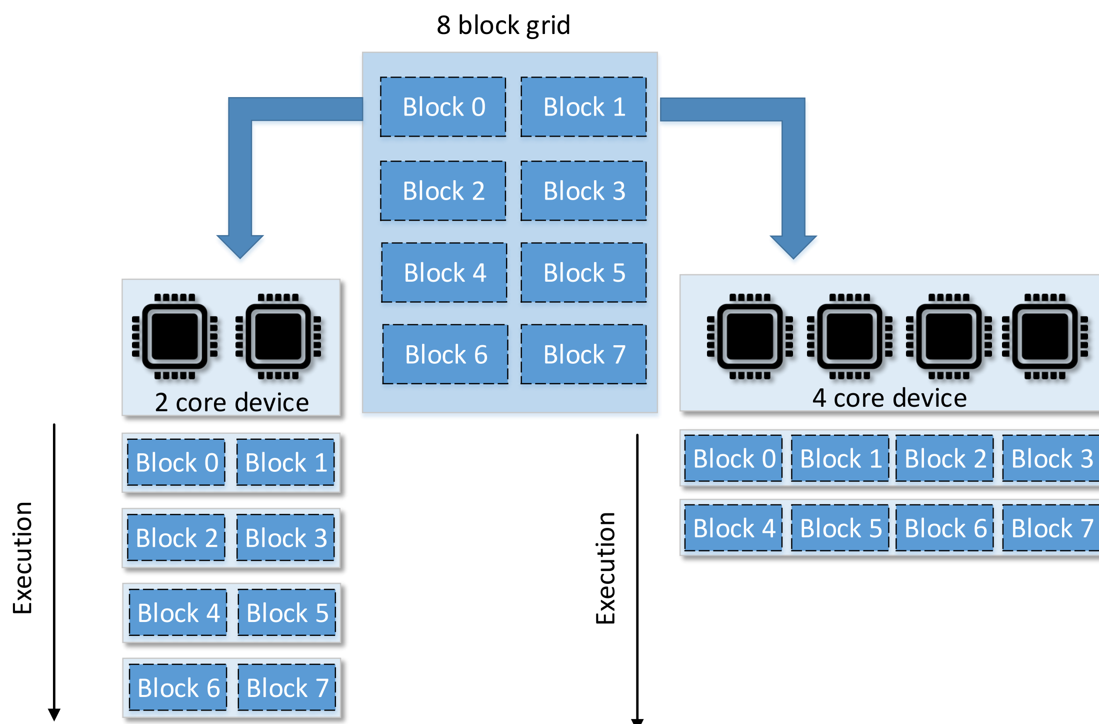


Figure 2.18. Scalability management [nVi12].

[2] The thread and block execution is nondeterministic.

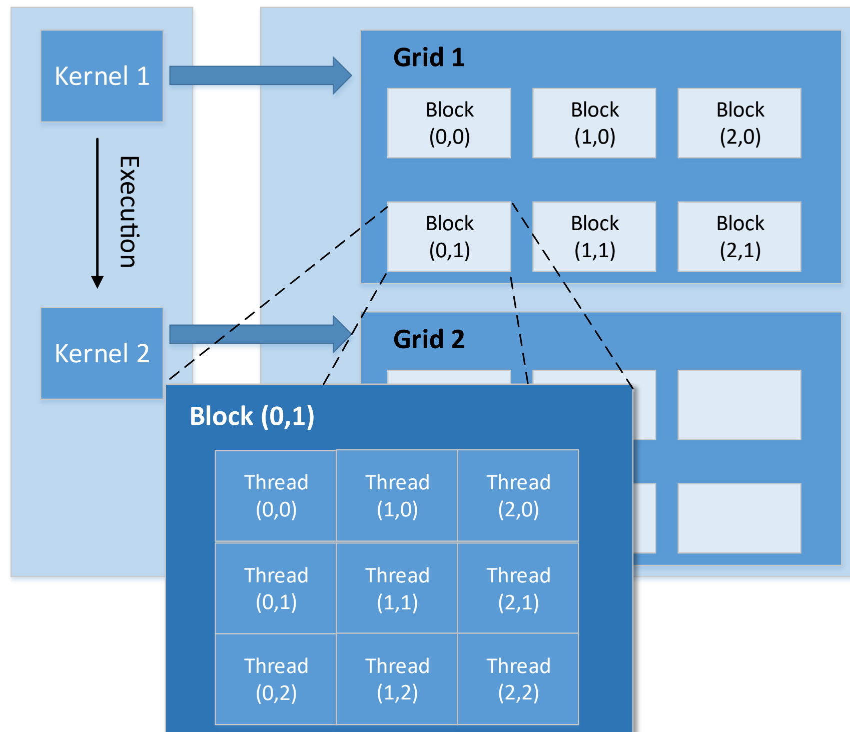


Figure 2.19. Example of thread and block distribution in two grids [nVi12].

2.5.2 CUDA in detail

As explained before, each thread and block has its own identifier. This way, a programmer can decide where each part of the data is processed. Similarly to the concept of the thread and block, CUDA provides the concept of the *grid*. While blocks can be unidimensional (1D) and bi-dimensional (2D), threads can be unidimensional, bi-dimensional and three-dimensional (3D). This approach simplifies memory addressing for multidimensional data, so it can be useful for image processing and complex mathematics problems involving matrices.

2.5.3 CUDA hardware outline

The CUDA GPU contains N^3 *streaming multiprocessors*, where each multiprocessor is a set of 8 *streaming processors (SPs)* (see Figure 2.20). Each streaming multiprocessor creates, plans and executes up to 24 *warps*[3] in one or more blocks (768 threads). Thus, each warp executes one instruction per 4 clock cycles [nVi12].

[3] A block is divided into 32-thread groups called *warps*.

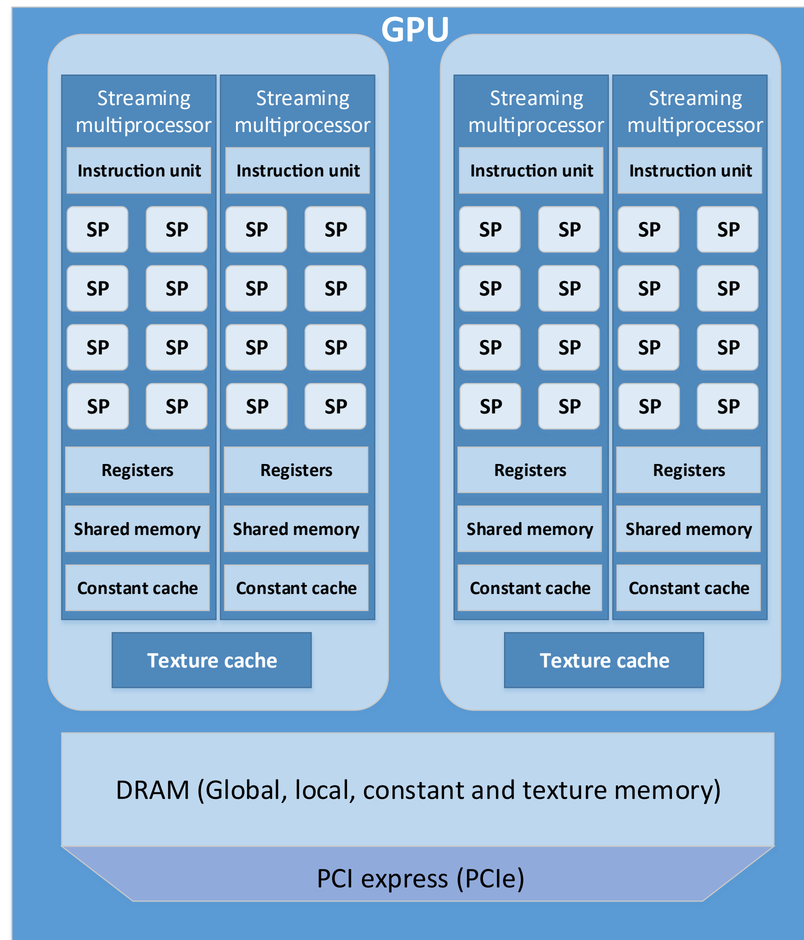


Figure 2.20. CUDA GPU hardware.

Streaming processors perform scalar operations over 32-bit single-precision integers and floats; similarly, each one executes independent threads, though all streaming processors should execute the instruction read by the *instruction unit* in every instant, according to the Single Instruction Multiple Thread (SIMT) framework. A set of threads executes the same instruction of the kernel in SIMT by exploiting data parallelism and to a lesser extent the tasks. Threads are managed by the hardware and are transparent to the programmer.

2.6 CUDA programming model

2.6.1 Introduction

CUDA programs are written in a C-like style language with an extended set of keywords for parallel programming specifics. As CUDA programming is C based, it is possible to integrate CUDA keywords with C/C++ language instructions in the same project.

As observed in Figure 2.21, the NVCC compiler discriminates between C/C++ code and CUDA-specific language. This process generates machine target code for the CPU and the Parallel Thread Execution (PTX) code that is similar to pseudointermediate code. Finally, the

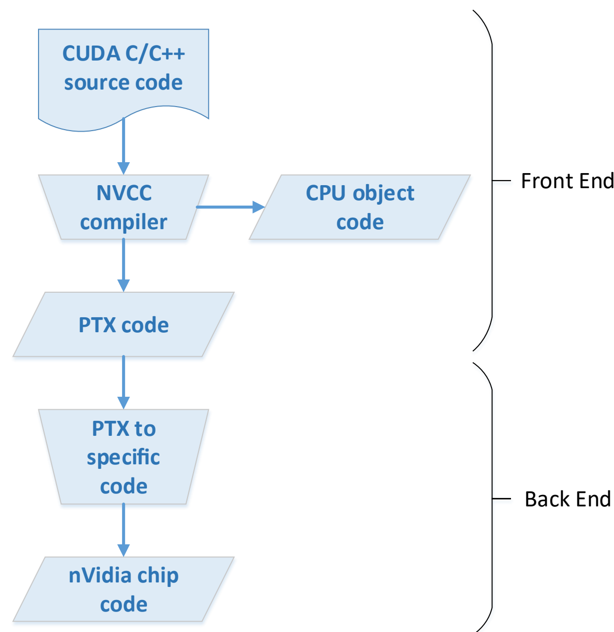


Figure 2.21. CUDA compilation phases.

PTX code is translated to the specific machine code of the nVidia chip.

2.6.2 Data level parallel programming

In the data parallelism model, the instructions are oriented to operate over a data set. This data is arranged either in arrays or cubes (three-dimensional arrays). This way, a group of tasks collectively work over different parts of the data structure. As can be appreciated from Figure 2.22 the main characteristic of this model is that tasks always apply the same instructions over different parts of the array. The origin of this model is the SIMD architecture and is the CUDA programming oriented technique.

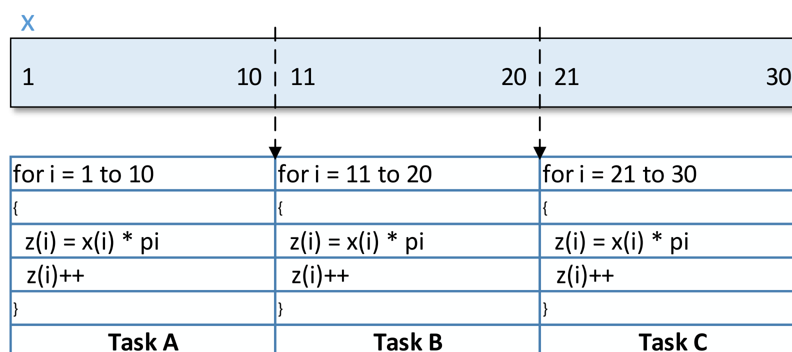


Figure 2.22. CUDA data level parallel programming example.

2.7 Cloud formation

According to Hans Häckel [Häc06], the air contains water but in vapour form. The water vapour is much more than «water in a gaseous state»; it is a absolutely invisible, colourless and odourless constitutive element. When air containing water vapour comes in contact with a sufficiently cold temperature, then small water droplets are formed in the natural process called *condensation*. However, the air can contain only a certain quantity of water vapour, the extreme condition of which is called the *saturation humidity*. The temperature at which condensation is produced is termed the *thermodynamic temperature of the dew point* or simply the *dew point*.

This previous phenomenon also happens in the atmosphere. In meteorology, the *free atmosphere* is understood as the region of the atmosphere not affected by the earth's surface. The microscopical droplets that appear suspended in the free atmosphere become visible in the form of clouds. However, a *surface tension* acts over them, which tends to shrink the droplet surface. This mechanism increases the tension inside the droplets and, as consequence, the water molecules experiment a surface push outward and are ultimately lost to the environment.

Usually, small dust particles contribute to cloud formation. There are untold numbers of these particles in the atmosphere. In pure air, there are approximately 100 particles per cm^3, but in polluted cities, there are normally 1 million per cm^3. Their diameter is approximately $1\mu m$, but smaller particles can also exist. In meteorology they are called *aerosols*.

Many of these particles, originating from cities, forest fires, artificial combustion, heaters, vehicle engines or volcanoes, contain small salt crystals. It is known that salt usually attracts water. These salt particles also allow the formation of clouds in the free atmosphere.

In physics, hot air is less heavy and tends to move upward, whereas cold air exhibits the opposite effect. When different parts of the ground warm up, such as roads in summer, the rising of isolated and self-contained air volumes are produced in the manner of air bags, called *air layers*. These volumes start to rise until they reach the *level of condensation* where a cloud is formed. Since the level of condensation can be considered a flat and horizontal surface, the lower limit of clouds appears as a smooth plane.

The blue and yellow lines in Figures 2.23 and 2.24 represent the decreasing temperature in relation to the altitude and the free atmosphere temperature according the altitude, respectively. In Figure 2.23 the cloud starts growing at a condensation level at 1.8 km of altitude (0.9 km in Figure 2.24). This process is called *expansion*. When the volume of hot air reaches 7 km of altitude as shown in Figure 2.23, it and the free atmosphere have the same temperature. Far beyond this altitude, a *thermal inversion* happens. This phenomenon occurs when the air volume has a temperature lower than the free atmosphere, and consequently clouds cannot form. A similar situation appears in Figure 2.24.

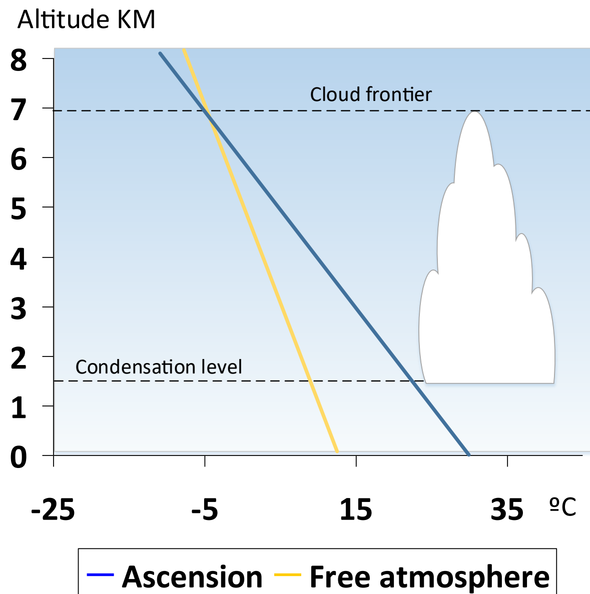


Figure 2.23. Vertical profile of a cloud.

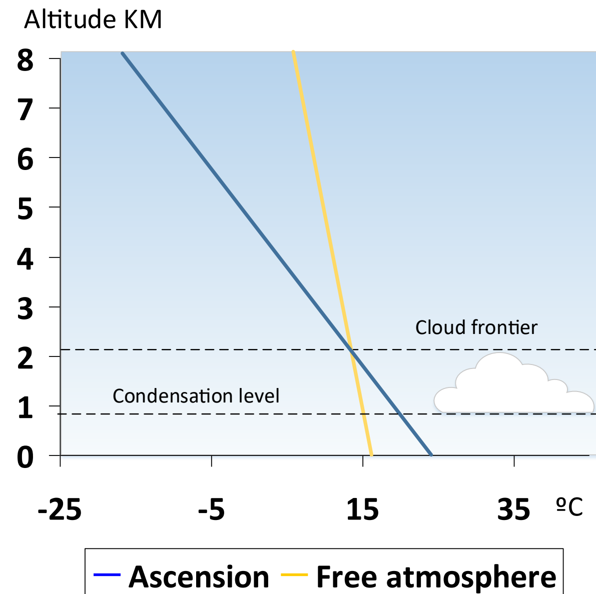


Figure 2.24. Flat cloud.

2.8 Cloud taxonomy

The rigorous classifications of clouds did not begin until the nineteenth century. Jean Baptiste Lamarck (1744-1829) and Luke Howard (1772-1864) independently published their works in 1801 and 1803, respectively. Howard was a prominent personality in the world of cloud classification. Moreover, his taxonomy has been improved over the years. Howard's classification was based in observable characteristics such as the height, extension and form. He also used Latin names for his sorting, which fit well in the scientific worldview of that age. Johann Wolfgang von Goethe was attracted by Howard's classification and dedicated several poems to them. Finally, the efforts of H. Hildebrandsson and R. Abercromby led to the first international atlas of clouds in 1896.

The actual cloud classification is guided by a continually revised publication edited by the World Meteorological Organization (WMO), an UN subdivision. The last revision was released in 1987.

According to Figure 2.25, the classification of the 10 genera of clouds by altitude is as follows:

2.8.1 High clouds

- **Cirrus (Ci):** Isolated clouds in the form of white and subtle filaments. They look like hair. They are formed by ice crystals at temperatures below -40°C.
- **Cirrocumulus (Cc):** Spots or bands of isolated small thin white clouds of granulated aspect without shadows. Their appearance is usually regular.
- **Cirrostratus (Cs):** White and transparent cloud veils of a fibrous or smooth aspect covering the sky in part or in total. They normally produce halo effects and are formed

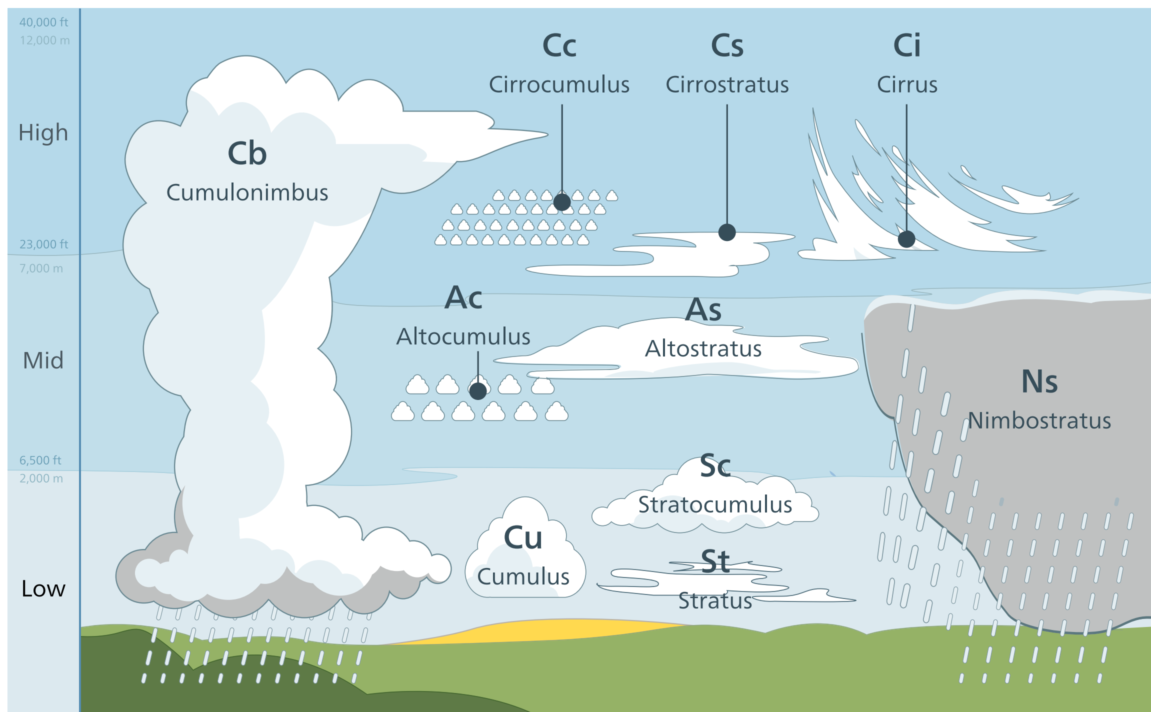


Figure 2.25. Cloud taxonomy.

by small ice crystals.

2.8.2 Middle clouds

- **Altocumulus (Ac):** Spots or bands of white or grey clouds in the form of sheets or rollers with a diffuse aspect. They can be isolated or not and are formed by a mixture of small ice crystals and supercooled water droplets.
- **Altostratus (As):** Bands or layers of grey or bluish clouds with a fluted, fibrous and uniform character. They usually cover the sky, allowing only glimpses of the sun. They are mixed clouds that allow iridescence and crown effect observation.

2.8.3 Low clouds

- **Stratocumulus (Sc):** Spots, bands or layers of grey and/or white clouds, with dark parts in the form of mosaics. They appear attached in fibrous shapes and contain small water droplets above 0°C.
- **Cumulus (Cu):** Isolated clouds, usually dense and well delimited. They normally grow in the vertical direction in the form of protuberances, domes or towers resembling a cauliflower. The lower side is dark and flat. The sun-illuminated parts are very radiant. The cumulus temperature lies above 0°C.
- **Stratus (St):** Totally grey cloudy layer with an uniform low frontier that may generate rain.

- **Cumulonimbus (Cb):** Large and dense clouds with vertical growths like a high mountain. They usually have flat or fibrous forms in the upper part. These clouds are frequently very dark under the lower limit. They may produce rain.
- **Nimbostratus (Ns):** Dark and grey clouds that imply rain or persistent snowfall. The sun cannot be seen through them. They usually generate the optical phenomenon called a rainbow.

[Häc06].

2.9 Meteorological concepts

This section examines several meteorological theorems and properties related to cloud physics. It is not the aim of this section to delve too deeply into complex concepts that may be outside the scope of the present thesis.

2.9.1 Atmospheric composition

Air is composed of several gases and other liquid/solid particles as explained in Section 2.7. The atmosphere is a mixture of nitrogen (78%), oxygen (21%) and argon (1%) and very small quantities of neon, helium and other gases in a permanent form. This composition is uniform up to 90 km of altitude. The most abundant gases in the atmosphere in variable concentrations are water vapour, carbon dioxide and ozone. A very important concept in meteorology is that of *moist air* which is a mixture of two ideal gases: dry air and water vapour [RY96].

2.9.2 Ideal gas

The ideal gas satisfies the following equation:

$$pV = RT \tag{2.13}$$

where p is the pressure, V is the molar volume, R is the universal gas constant and T is the temperature.

If we assume the mass of one mole as M, then the density $\rho = M/V$. Therefore, if we substitute these relations into Equation 2.13, we obtain:

$$p = R_c T \rho \tag{2.14}$$

where the constant $R_c \equiv R/M$ [And10].

2.9.3 First Law of Thermodynamics

As explained by Andrews [And10], the First Law of Thermodynamics as applied to a closed volume is expressed mathematically as:

$$dU = dQ + dW \tag{2.15}$$

where dU is the amount of internal energy, dQ is the heat in the system and dW is the work performed.

If we assume functions of state, then Equation 2.15 can be formulated as:

$$dU = TdS - pdV \tag{2.16}$$

where S is the *entropy*. Equation 2.16 can be rewritten as:

$$dH = TdS + Vdp \tag{2.17}$$

where $H = U + pV$ is the *enthalpy*.

Equation 2.14 implies that for a unit mass of air $(V = 1/\rho)$ and $U = c_v T$, where c_v is the specific heat capacity at constant volume:

$$H = c_v T + R_c T = c_p T \tag{2.18}$$

where c_p is the specific heat at constant pressure. If Equation 2.18 is substituted into Equation 2.17 and divided by T, we deduce:

$$S = \int c_p d(\ln T) - R_c d(\ln p) \tag{2.19}$$

By integrating Equation 2.19, we finally obtain:

$$S = c_p \ln(T p^{-\kappa}) + C \tag{2.20}$$

where $\kappa = R_c/c_p \equiv 2/7$ for a diatomic gas and C is a constant.

The *potential temperature* (θ)[45] of a mass of air is derived from Equation 2.19 by assuming $dS = 0$.

$$\theta = T\left(\frac{100kPa}{p}\right)^{\kappa} \tag{2.21}$$

where T is the temperature of the air at pressure p.

[4] An *adiathermal* process occurs when heat is neither gained nor lost.
[5] An *adiabatic* process is both adiathermal and reversible.

2.9.4 Water vapour state equation

When the water is in the vapour phase, it has an ideal gas behaviour, and its state equation is:

$$e = \rho_v R_v T \tag{2.22}$$

where e is the vapour pressure, ρ_v is the vapour density and R_v is the ideal gas constant equal to $461.5 J kg^{-1} K^{-1}$ [RY96].

2.9.5 Hydrostatic equilibrium

As cited by [RY96], the air is balanced because the vertical pressure force on the air equilibrates to the force of gravity as seen in Figure 2.26.

Therefore, the hydrostatic equation is stated as:

$$\frac{\partial p}{\partial z} = -\rho g \tag{2.23}$$

where p is the pressure at height z.

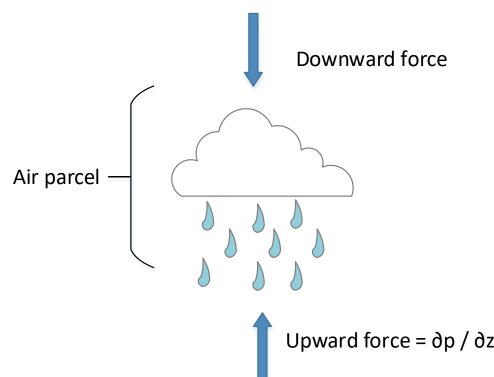


Figure 2.26. Hydrostatic equilibrium applied to a parcel of air.

2.9.6 Dry adiabatic lapse rate

The dry adiabatic lapse rate for moist air is denoted by the equation:

$$\frac{dT}{dz} = -\frac{g}{c_p} = -\Gamma \tag{2.24}$$

where $\Gamma = 0.98$ °C/100 m $\equiv 10^{-2} K/m$

"This is the rate at which temperature falls off with height in the process of dry adiabatic ascent" [RY96].

2.9.7 Buoyant force

Consider a parcel of dry air with volume V, temperature T and density ρ. This parcel displaces an equal volume of air of temperature T' and density ρ'. We assume that the

downward force on the parcel and the displaced air is $\rho g V$ and $\rho' g V$, respectively, whereas the upward force is the same for both the parcel and the displaced air and is defined by $-V\frac{\partial p}{\partial z}$. If the net buoyant force (upward) is $Vg(\rho' - \rho)$, the mass per unit is:

$$F_B = g\frac{\rho' - \rho}{\rho} = g\frac{T - T'}{T'} \tag{2.25}$$

As a conclusion, the force is positive when the parcel is warmer than the ambient air, and vice versa [RY96].

2.9.8 Stability criteria of dry air

Equation 2.24 of the dry adiabatic lapse rate will serve us to evaluate the stability of atmospheric layers with respect to the vertical displacement of an air parcel. As is seen here, the stability condition depends on the ambient lapse rate.

Taking T as the ambient air parcel temperature and Δz as the adiabatically raised distance that is cooled by the amount $\Gamma\Delta z$, then the temperature is decreased to $T - \Gamma\Delta z$.

$$\gamma = -\left(\frac{\partial T}{\partial z}\right) \tag{2.26}$$

Equation 2.26 defines the ambient lapse rate. Hence, the excess temperature over that of the ambient air is $\Delta z(\gamma - \Gamma)$. When this expression is positive, the ambient air parcel is warmer than the surroundings parts, and by the effect of the buoyant force, 2.25 predicts movement upwards.

As a conclusion, the stability criteria for dry air is summarized in the following cases:

- $\gamma < \Gamma$ Stable
- $\gamma = \Gamma$ Neutral
- $\gamma > \Gamma$ Unstable

[RY96].

2.9.9 Stability criteria for moist air

Consider the mathematical expression of the pseudoadiabatic process [RY96]:

$$\frac{dT}{T} = k\frac{dp}{p} - \frac{L}{Tc_p}dw_s \tag{2.27}$$

where w_s represents kilograms of water vapour.

We proceed to differentiate Equation 2.27 with respect to height:

$$\frac{dT}{dz} = \frac{kT}{p}\frac{dp}{dz} - \frac{L}{c_p}\frac{dw_s}{dz} \tag{2.28}$$

and employing Equation 2.23 on the hydrostatic response and the Clausius-Clapeyron equation, we obtain the *pseudoadiabatic lapse rate*:

$$\Gamma_s = -\frac{dT}{dz} = \Gamma \frac{\left(1+\frac{Lw_s}{R'T}\right)}{\left(1+\frac{L^2 \varepsilon w_s}{R' c_p T^2}\right)} \quad (2.29)$$

As a consequence, the moist air stability criteria is:

- $\gamma < \Gamma_s$ Absolutely stable
- $\gamma = \Gamma_s$ Saturated neutral
- $\Gamma_s < \gamma < \Gamma$ Conditionally unstable
- $\gamma = \Gamma$ Dry neutral
- $\gamma > \Gamma$ Absolutely unstable

2.10 Radiometry

Radiometry is the branch of physics that studies the interaction of light radiation with matter. It is useful to calculate the effect of light on the condensed water vapour inside and outside the cloud. The history of light as a concept began in Ancient Greece and, in the Middle Ages, with the description of reflection and refraction phenomenon by Al-Hazen (eleventh century). Later, in the seventeenth century, the first experimental laws were described by Descartes and Snell. At the end of the seventeenth century, Huygens and Newton announced their opposing theories called undulatory and corpuscular theory, respectively. Newton's theory prevailed until 1801 when Young, Fresnel and Foucault recovered Huygens' undulatory theory as well.

2.10.1 Main radiometric cloud phenomena

It is well known that light consists of electromagnetic radiation that the human eye can perceive. The visible spectrum ranges from 380 to 740 nanometres (430-770 THz). The frequency (ν), the period (T) and wavelength (λ) are related by the following equation:

$$c = \lambda \nu = \frac{\lambda}{T} \quad (2.30)$$

where c is the propagation velocity in the medium, $3 \cdot 10^8 m \cdot s^{-1}$ in a vacuum and lower in other media.

The nine major radiometric phenomena related to clouds in this thesis are explained below:

- **Absorption:** This phenomenon occurs when the light radiance is transformed into energy, specifically heat, when it interacts with other objects. For example, when light impacts on a perfect grey surface, 50% of the radiation is reflected, but the remainder is converted into heat.

- **Extinction:** This phenomenon is related to the attenuation of light due to absorption when traversing a medium. If we define K_s as the scattering coefficient and K_a as the absorption coefficient, the extinction coefficient is determined as $K = K_s + K_a$ [Har03].

- **Optical depth:** This effect is a mathematical concept that measures the relative loss of light when traversing a medium of thickness s, where 1 represents the maximum opacity as expressed in Equation 2.31.

$$\tau(0,s) = \int_0^s K(\vec{x} + t\vec{\omega})dt \quad (2.31)$$

where $\vec{x}$ is a point inside the medium and $\vec{\omega}$ is the direction of light propagation [Har03].

- **Transmittance:** This mathematical concept is related to the amount of light transmitted from a volume unit to an adjacent volume unit. It is expressed in Equation 2.32.

$$T(s,s') = e^{-\tau(s,s')} \quad (2.32)$$

As a consequence, the opacity of the integral is defined as $\alpha(s,s') = 1 - T(s,s')$ [Har03].

- **Absorbance:** The absorbance is the ratio between the light intensity transmitted by the material and the light intensity received by the material.

$$A = -log_{10}\left(\frac{I_{output}}{I_{input}}\right) = -log_{10}T \quad (2.33)$$

- **Scattering:** This term refers to the diffusion of light while traversing a medium in multiple directions other than the direction of the light source.

- **Single scattering:** Here, most of the light is scattered in the direction of incidence. This phenomenon occurs when light traverses small particles or very transparent materials. One example is the interaction of light with the condensed droplets of a container of beer.

- **Multiple scattering:** In this case, the light is scattered in multiple directions and reaches multiple particles, similar to a chain reaction. This effect occurs when light traverses a collection of small particles such as condensed water vapour.

- **Scattering albedo:** The single scattering albedo is the proportion of attenuation by extinction $\sigma = \frac{K_s}{K}$. «Single scattering albedo is the probability that a photon "survives" an interaction with a medium» [Har03]. The albedo ranges from 0 to 1, where 0 means no scattering and 1 no absorption.

2.10.2 Phase functions

The phase functions are probability formulas that express the amount of light from the incident direction ω that is scattered into another direction ω'. Scattering depends on the phase angle ϕ between the incident light direction ω and resultant light direction ω' as illustrated in Figure 2.27.

The phase function is dimensionless, and its probability distribution adds up to 1 when integrated over the entire solid angle.

$$F_x(x) = \int_{4\pi} P(\omega,\omega')d\omega = 1 \tag{2.34}$$

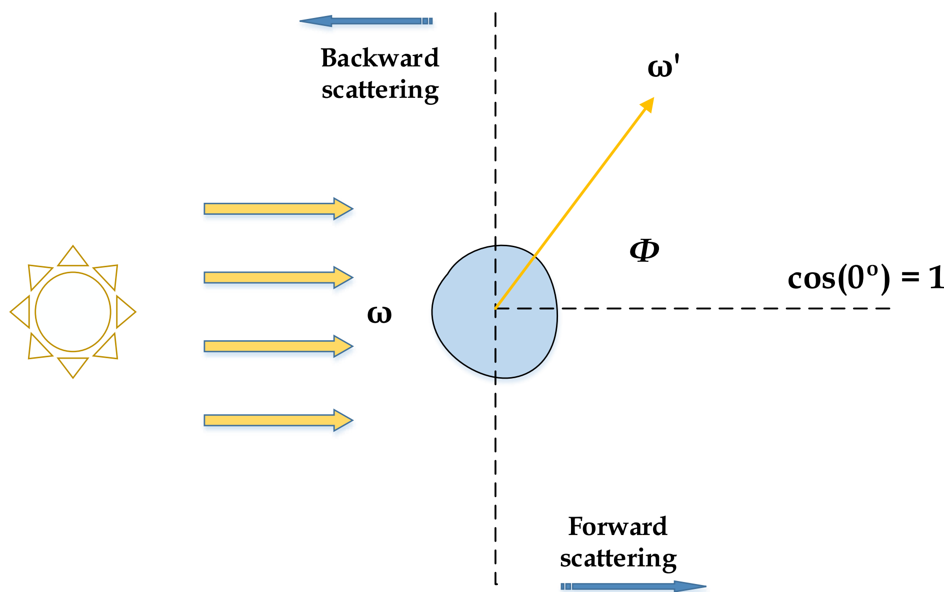


Figure 2.27. Scattering angle, ϕ, represented as the relation between the incident light from the sun and scattered light from the water droplet.

There are several types of phase functions; typical examples are discussed in the following paragraphs:

- **Rayleigh phase function:** This phase function was proposed by John William Strutt (Lord Rayleigh) and models the scattering of light by particles in the atmosphere to produce the blue sky phenomenon.

$$P(\theta) = \frac{3}{4}\frac{1+cos^2(\theta)}{\lambda^4} \tag{2.35}$$

  As indicated in Equation 2.35, θ is the angle between ω and ω', and λ is the wavelength of the incident light. The squared $cos(\theta)$ gives more weight to the directions closer to the scattering limit values.

- **Henyey-Greenstein phase function:** This phase function was developed by Louis G.

Henyey and Jesse L. Greenstein in 1941 and it is a useful model of Gustav Mie's complex scattering theory.

$$P(\theta) = \frac{1}{4\pi}\frac{1-g^2}{[1+g^2-2gcos(\theta)]^{3/2}} \tag{2.36}$$

where g is the symmetry parameter and ranges within $-1 \le g \le 1$ with the following meaning:

$$g = \begin{cases} < 0 & \text{backscattering (most of the light is scattered back to the light source)} \\ 0 & \text{isotropic scattering (no scattering)} \\ > 0 & \text{forward scattering (most of the light is scattered in the incident direction)} \end{cases} \tag{2.37}$$

As seen in Equation 2.36, the function works on a ellipse, where its values are greater near the poles.

Figure 2.28 is a simulation of the scattered light intensity as a function of the angle using a particle system; Figure 2.29 is the same simulation of light scattering but using a polar representation. Both figures were plotted with the MiePlot application [Lav17].

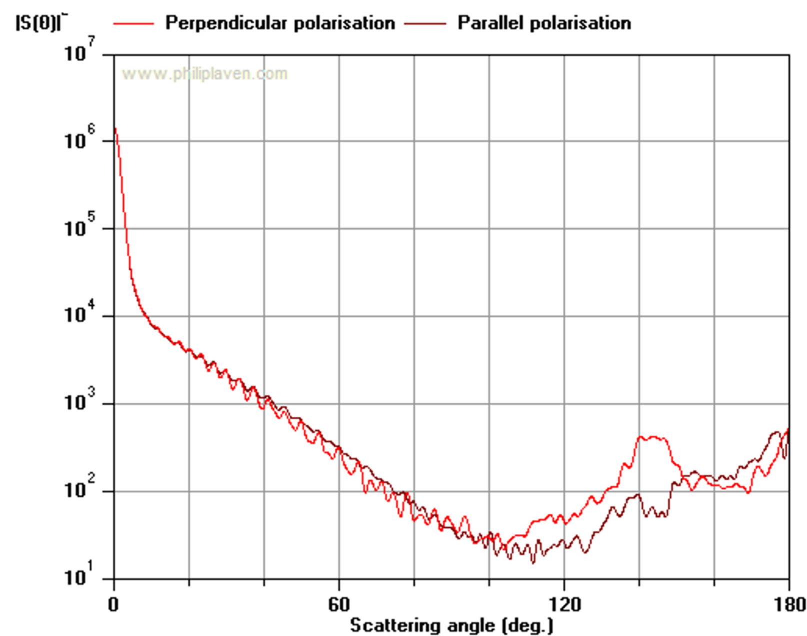


Figure 2.28. Plot of the Mie scattering angle per light intensity using 50 particles for a cumulus distribution with perpendicular and parallel polarisations.

A basic plot in Figure 2.30 of the Henyey-Greenstein phase function can help to illustrate the previous concepts:

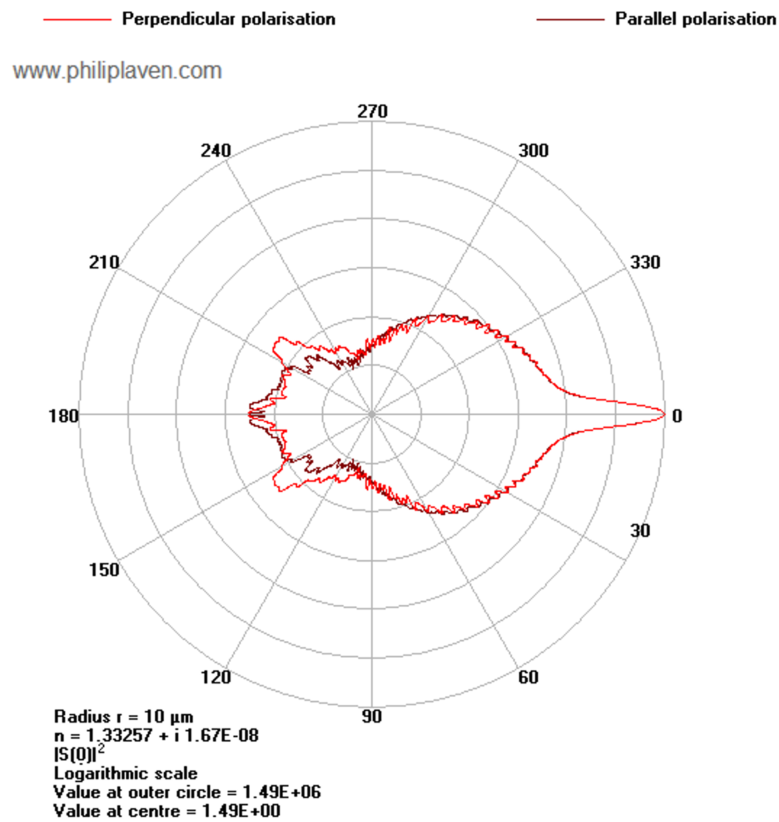


Figure 2.29. Polar plot of the Mie scattering in a water droplet with 50 particles for a cumulus distribution and both polarisations. A high forward scattering appears in the image.

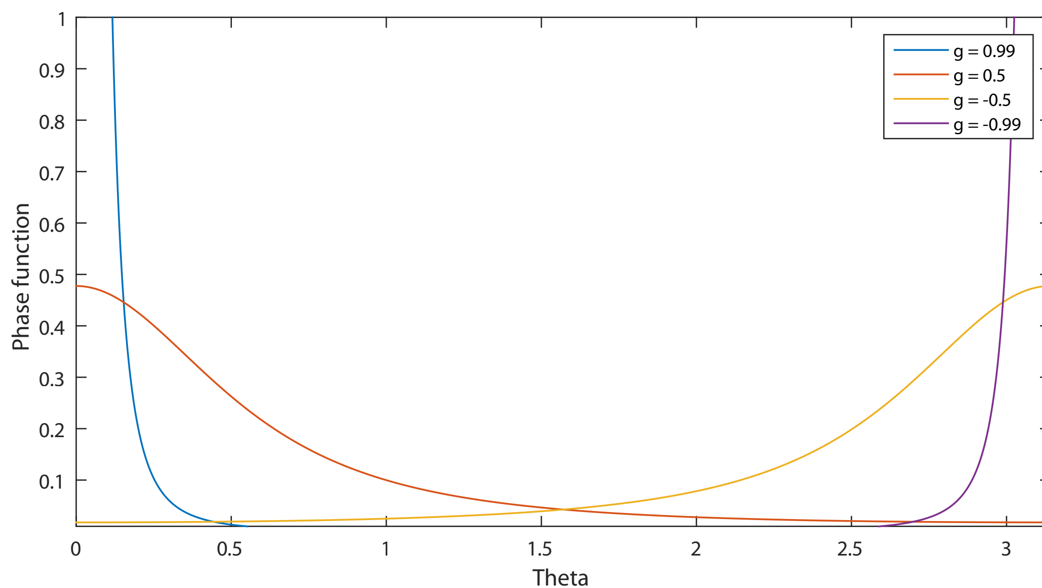


Figure 2.30. Henyey-Greenstein phase function for asymmetry parameters g = 0.99, g = 0.5, g = -0.5 and g = -0.99.

2.10.3 Light transport model

The mathematics of the light transport model applied in this thesis is much like that based on the work by Mark Harris and Nelson Max[Har03; Max95] as explained in this section.

As seen in Figure 2.31, the intensity of light coming from the light source is attenuated due to absorption and outscattering, while the inscattering intensifies the incoming light.

Letting L be the intensity of the light and K_a the absorption coefficient, we define the *Absorption* term as:

$$\frac{dL(\vec{x},\vec{\omega})}{ds} = -K_a(\vec{x})L(\vec{x},\omega) \tag{2.38}$$

where Equation 2.38 expresses the change in light intensity caused by the absorption over distance.

The *outscattering* term is written in a similar form:

$$\frac{dL(\vec{x},\vec{\omega})}{ds} = -K_s(\vec{x})L(\vec{x},\omega) \tag{2.39}$$

where Equation 2.39 is the variation of the light intensity as affected by the distance in the $\vec{\omega}$ direction.

Therefore, as cited in Subsection 2.10.1 on the K coefficient, we can combine Equations 2.38 and 2.39 to represent absorption and outscattering in the same expression:

$$\frac{dL(\vec{x},\vec{\omega})}{ds} = -K(\vec{x})L(\vec{x},\omega) \tag{2.40}$$

Regarding the *inscattering* term, we need to calculate all the incoming scattered light in all solid angle directions (4π) by taking into account the scattering directions:

$$\frac{dL(\vec{x},\vec{\omega})}{ds} = K_s(\vec{x}) \int_{4\pi} P(\vec{x},\vec{\omega},\vec{\omega}')L(\vec{x},\omega')d\omega' \tag{2.41}$$

Finally, the *light transport* formula is a single differential equation that combines Equation 2.40 and Equation 2.41 as seen below:

$$\frac{dL(\vec{x},\vec{\omega})}{ds} = -K(\vec{x})L(\vec{x},\vec{\omega}) + K_s(\vec{x}) \int_{4\pi} P(\vec{x},\vec{\omega},\vec{\omega}')L(\vec{x},\omega')d\omega' \tag{2.42}$$

Equation 2.42 is approximated with the assumption of Max [Max95] and Harris [Har03] and used in this thesis for volumetric cloud lighting as seen in Chapter 5.

$$L(D,\vec{\omega}) = \underbrace{L(0,\vec{\omega})T(0,D)}_{\text{transmission}} + \underbrace{\int_0^D g(s)T(s,D)ds}_{\text{in-scattering}} \tag{2.43}$$

where

$$g(s) = K_s(\vec{x}(s)) \int_{4\pi} P(\vec{x},\vec{\omega},\vec{\omega}')L(\vec{x}(s),\omega')d\omega' \tag{2.44}$$

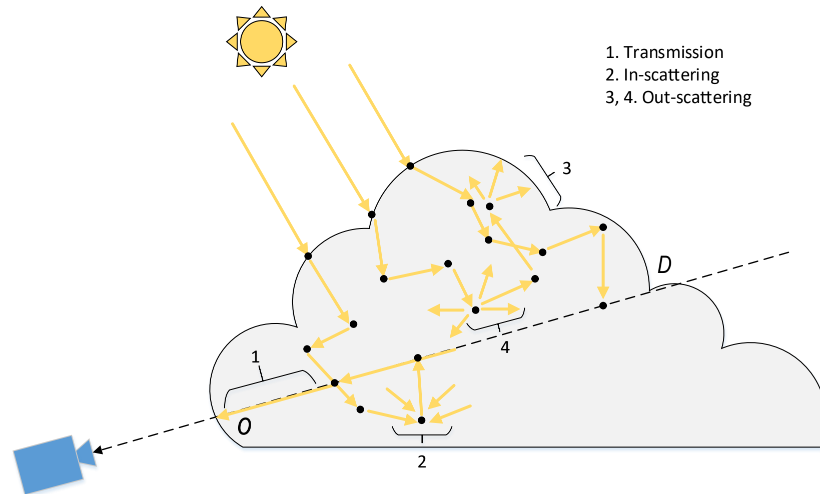


Figure 2.31. The effect of inscattering, outscattering, transmission and attenuation in the light transport model.

Because this is a multiple scattering light transport model, it generates a problem of recursion as expressed in Equation 2.43. This thesis addresses this issue in Chapter 5.

2.11 Fluid dynamics

Cloud advection and convection depend of the fluid state of the gas mixture in the atmosphere. Thus, advection is treated as the cloud movement caused by the velocity of the fluid, whereas convection is related to the fluid motion due to thermal states in the cloud. The equations of cloud dynamics can be expressed via either the *Euler equations of an incompressible fluid* or the *Navier-Stokes equations of incompressible flow*. The second option is selected for this thesis; a basic demonstration of the Navier-Stokes equation for a compressible fluid is presented here, as most cases in real life are based on compressible fluids.

2.11.1 The Navier-Stokes equation

The explanation of the Navier-Stokes equation is based on [And10], which reports that the equation is developed based on Newton's Second Law:

$$F = m \cdot a \implies (\rho \delta V)a = \delta F \tag{2.45}$$

where $\delta V = \delta x \delta y \delta z$ is the volume of a cubical water droplet, $\rho \delta V$ is the mass, a is the acceleration and δF is the force vector.

If we consider the pressure force acting in the x direction of the cubic droplet, the force at position x is $p(x)\delta A$ in the positive abscissa, where $\delta A = \delta y \delta z$ is the area of the droplet. Therefore, the pressure in the negative abscissa direction at position $x + \delta x$ is:

$$p(x+\delta x)\delta A \approx \left(p(x) + \frac{\partial p}{\partial x}\delta x + ... \right) \delta A \tag{2.46}$$

by using a Taylor expansion formula.

In the positive abscissa direction, the equation remains:

$$-\delta x \frac{\partial p}{\partial x} \delta A = -\delta V \frac{\partial p}{\partial x} \tag{2.47}$$

Therefore, in three dimensions, the force is:

$$\delta F_{press} = -(\delta V)(\nabla p) \tag{2.48}$$

If the gravity force is considered to act downwards:

$$\delta F_{grav} = -(\rho \delta V) g \vec{k}. \tag{2.49}$$

where $\vec{k} = (0,0,1)$.

Finally, we consider the viscous forces applied on the droplet as seen in Figure 2.32, where the flow is in the abscissa direction and varies only in the z direction. From the kinetic theory of gases, the x component of the viscous force applied on a surface from the top is:

$$\tau = \eta \frac{du}{dz} \tag{2.50}$$

where η is the viscosity constant.

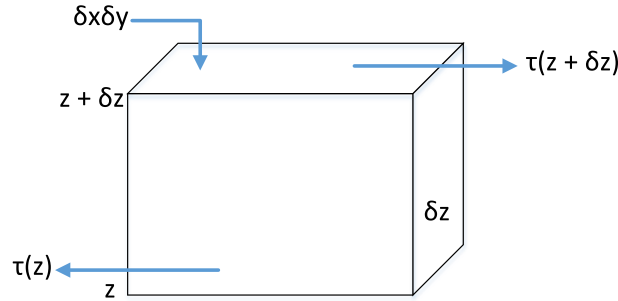


Figure 2.32. Horizontal forces applied on the bottom and top of the cubical droplet.

The net contribution to the viscous force is defined as:

$$[\tau(z+\delta z) - \tau(z)]\, \delta x \delta y \approx \frac{\partial \tau}{\partial z} \delta x \delta y \delta z = \frac{\partial \tau}{\partial z} \delta V \tag{2.51}$$

By using Equation 2.50, the following is obtained:

$$\eta \frac{d^2 u}{dz^2} \delta V \tag{2.52}$$

Thus, by generalizing the Equation 2.52, the viscous vector force is:

$$\delta F_{visc} = \delta V \eta \left(\nabla^2 u + \frac{1}{3} \nabla (\nabla \cdot u) \right) \tag{2.53}$$

if we consider the incompressible fluid for which $\nabla \cdot u = 0$.

Finally, by substituting Equations 2.48, 2.49 and 2.53 into Newton's Second Law and applying Equation 2.55 of a trajectory $(r(t))$ of a small droplet at a time t:

$$a = \frac{d^2r}{dt^2} = \frac{\partial u}{\partial t} + (u \cdot \nabla)u = \frac{Du}{Dt} \tag{2.54}$$

we achieve the Navier-Stokes equation for a compressible fluid as:

$$\frac{Du}{Dt} = -\frac{1}{\rho}\nabla p - gk + F_{visc} \tag{2.55}$$

where $F_{visc} = \delta F_{visc}/\delta V$.

2.12 Summary

This chapter is a brief review about the computer graphics methods and hardware technology related to efficient cloud rendering from a volumetric rendering perspective. The utilization of new multicore devices is an advantage for a state-of-the-art cloud display, along with the optimizations of large fluid grid structures benefiting from CUDA parallel programming. All these elements have been thoroughly applied in the present thesis and will be discussed in the forthcoming chapters. In addition to introducing the basic concepts related to cloud physics, a review of cloud formation conditions and cloud taxonomy is presented. In addition, an abbreviated explanation of general meteorological theory regarding to the atmosphere, thermodynamics and other related important concepts is presented. Because one of the most important aspects of this thesis is cloud lighting in all daytimes and circumstances, a concise overview of radiometry with the most relevant issues is developed. Finally, this part concludes by explaining the basic background for the understanding of the Navier-Stokes equations by illustrating the compressible fluid flow deduction as a basis for the incompressible assumption that will be briefly explained in Chapter 7.

CHAPTER 3

State of the art and author contributions

THis chapter deals with the relevant works about cloud rendering from the emergence of computer graphics to the latest findings. To this end, a brief definition about the two main branches in cloud generation is presented in Section 3.1. A complete overview of the state-of-the-art cloud rendering techniques will be developed and explained with several image samples in Section 3.2. Following that, the specific contributions of the author of this thesis relative to the previously mentioned methods are listed in Section 3.3. Finally, an author-developed comparison and the pros and cons of each feature per method are analysed using tables in Section 3.4.

3.1 Ontogenetics vs. physically based methods

Researchers studying computer-generated clouds have developed different approaches to addressing natural phenomena. Huang et al.[Hua+08] report on the two main groups of cloud rendering methods: *physically based models* and *ontogenetic models*. In the first approach, the atmospheric equations of cloud advection, thermodynamics, radiometry, and fluid dynamics are applied [Miy+01; Har+05]. In the second method, only a formal geometric visualization of the cloud in an artistic simulation using minimal physical characteristics is used.

While the physically based methods may be either more realistic but slower or less realistic but faster, the ontogenetic methods are a very useful choice to provide cloud rendering in many industrial and academic environments not needing very accurate scientific systems. This approach is an interesting and pragmatic option for the domestic world. The computer game industry requires more realistic but efficient cloud landscapes in its adventures. In contrast, the film industry prefers very-slow-to-process but accurate frames for their aesthetic value in story telling, whereas the computer games industry opts for balanced systems, i.e., a favourable middle point between realism and performance. However, in the context of Moore's law, the

objective trend in a future will be more physically based methods for all systems.

3.2 Cloud rendering methods

3.2.1 Texturized primitives

With this option, basic surfaces like skydomes or skyboxes are texturized with an omnidirectional true colour capture of the sky. Other implementations use procedural textures applied to spheres and ellipsoids; this was the starting point for cloud generation in the early years with the work of Gardner [Gar85] and Max [Max94]. This method may be considered as obsolete in modern GPUs, however, it is used in small devices that do not have three-dimensional (3D) acceleration. A state-of-the-art contribution to this approach was recently provided by Mukina and Bezgodov [MB15]. The main limitation of this method is the inability to approach, manoeuvre around, or traverse gaseous bodies, in addition to the lack of animation of the different cloud parts. This method is not used in this thesis due to the aforementioned limitations.

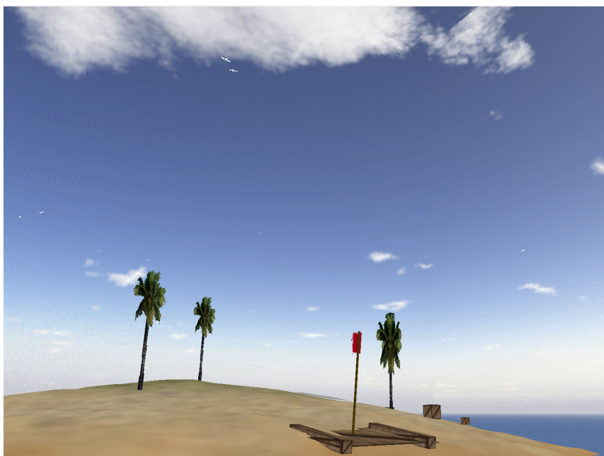


Figure 3.1. Virtual island [Jim03]. Example of a texturized skybox.

Figure 3.2. Procedural texturized skydome [Bek19].

As seen in Figure 3.1 the cloud representation in the island landscape has good realism; however, the clouds are drawn on the back side of a cube surface, thus avoiding any kind of interaction or animation. In addition to this drawback, a joining effect where the sides of the cube meet is produced.

Figure 3.2 is a sample of a rendered sky with the Lumion 3D application using a method similar to that of [MB15]. Although the clouds are procedurally generated with an artificial intelligence (AI) method, they lack three-dimensional aspects in the context of approaching, manoeuvring around or traversing.

3.2.2 Particle systems

The particle system method [Ree83] is based on a very basic and small three-dimensional geometry with an applied transparent texture. The typical three-dimensional geometries include triangles, quads and simple polygons. This technique is used for creating fire, explosions, grass, waterfalls, stars, rain, snow and smoke effects. A particle system structure (Figure 3.3) contains a set of tiny objects that emanate from a container called an *emitter*. The emitter controls the particle position and speed. Each particle has a lifespan, colour, texture and shape. In addition to these elements, other objects control the particle behaviour as affected by the gravity, wind and so on. Since 2002 authors have tried to achieve the simulation of clouds using basic quads or triangle surfaces with a Gaussian texture [Hua+08]. Harris conceived a new algorithm using particles with an efficient multiple scattering illumination system, which was based on Max equations [Max95]. The resulting rendering was improved with the use of impostors [Har+05; HL01; Har02]. The use of this technology increases performance and speed. This technique is considered useful for physical workload model simulators. However, the overall realism is not accurate enough as seen in Figures 3.4 and 3.5.

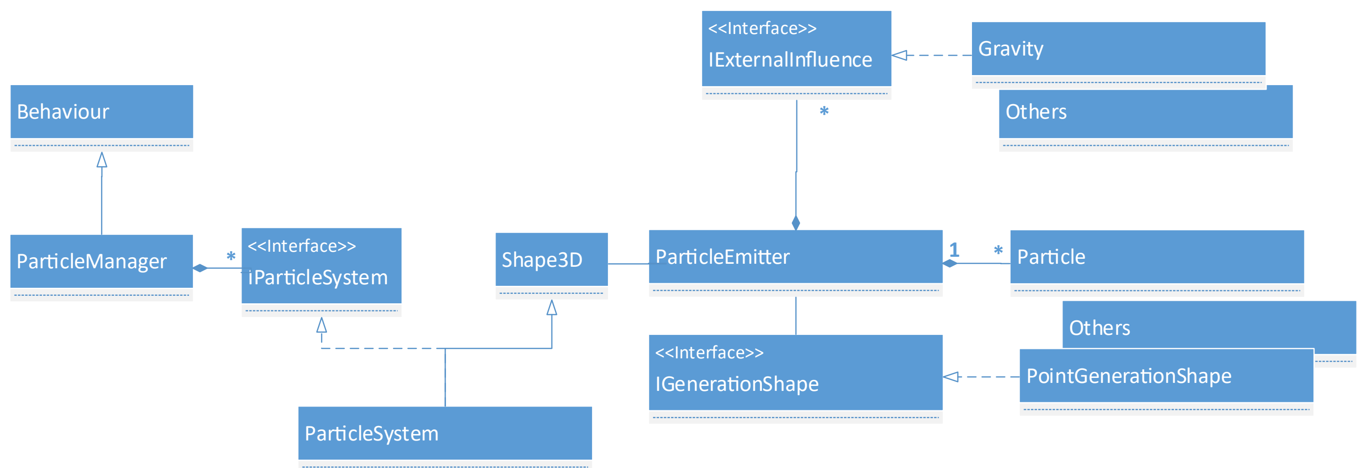


Figure 3.3. UML class diagram of a particle system [Jac08].

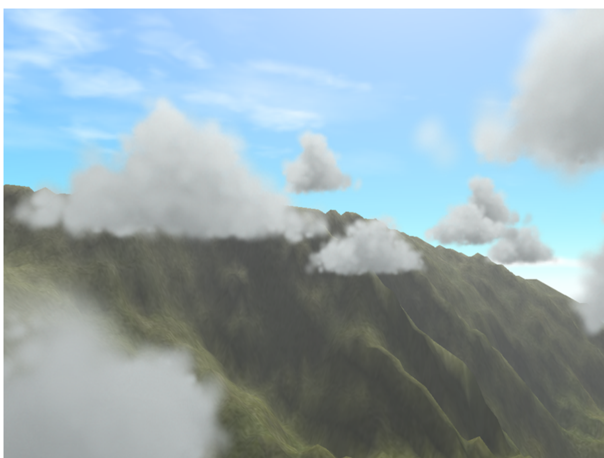


Figure 3.4. Particle system cloud using the [Har03] method.

Figure 3.5. Particle system cloud using the [Hua+08] method.

3.2.3 Geometry distortion

The basic idea behind this technique by [Kni+02] consists in drawing a complex cloud mesh with a group of megaparticles as explained by Ebert [Ebe97] and Engel [Eng07]. Then, a lighting algorithm, like *Phong* or *Gouraud* shading, is used on the geometry. After this step, the cloud map needs blurring using a *Gaussian filter* to distort it. To do this, a quad is placed at the center of every cloud and billboard vertex with respect to the camera, covering the entire cloud from any angle. This quad is rendered to distort the cloud map and sample a 2-channel fractal/noise texture to obtain a distortion offset. Afterwards, the blurred cloud map is sampled using texture coordinates distorted by this offset and the distance from the quad to the camera. Optionally, a radial blur may be performed to soften the resulting image. At a final stage, the render target is merged to the back buffer. This method is midway between particle systems and volumetric rendering, with good performance and easy shading, but it lacks accurate realism and is not suitable for cloud shapes other than cumulus. This method has not been used in this thesis because it is difficult to adapt for arbitrary-shaped clouds.

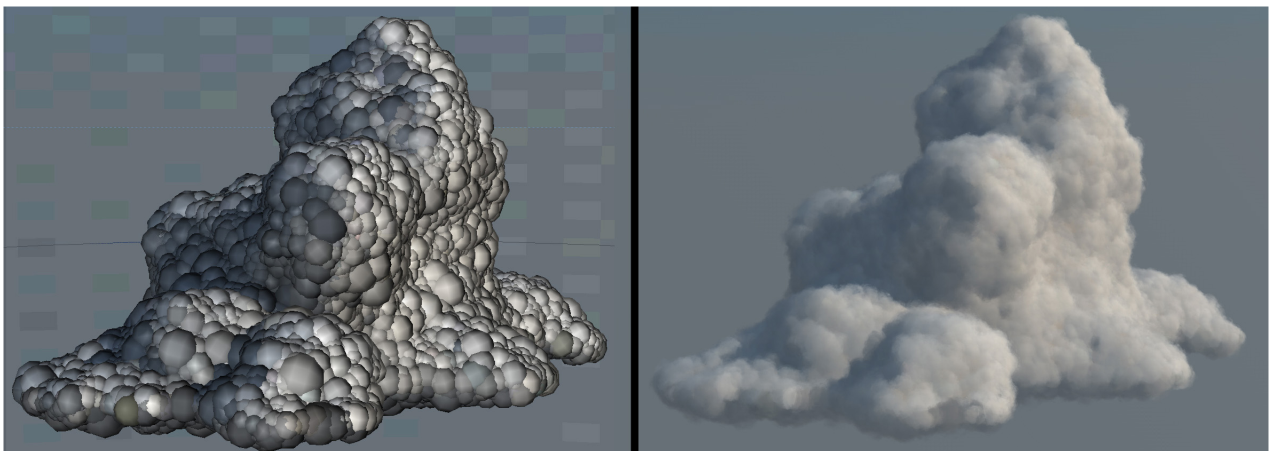


Figure 3.6. A cumulus cloud produced by distorting a set of spheres [Gir12].

As can be appreciated in Figure 3.6, the complete set of sphere-based mega-particle-generated shapes is grouped in a cumuliform cloud distribution. This approach allows the application of a basic but efficient lighting system as mentioned before. The results have good quality, and the shadowing and occlusion effects are easy to implement for cumulus types in most of the cases. The animation and dynamics can be obtained by controlling the sphere centres in a particle-system-like style.

3.2.4 Volumetric rendering

This model represents the current state-of-the-art in cloud rendering, and as such it is used in computer games, flight simulations, and real-time computer graphics. The main references for volumetric rendering are found in [DCH88; Lev88]. Volumetric rendering is the model upon which the research work explained in this thesis is based. This technique is the same as that used to obtain two-dimensional (2D) medical images from 3D magnetic resonance or computed tomography data. Volumetric rendering of clouds still implies a heavy workload for most modern GPUs, but the realism is very accurate and may become standard in the coming years. Later contributions to this approach were made by Yusov [Yus14], Elek et al. [Ele+12], Dobashi et al. [Dob+00], and Goswami-Neyret [GN17], while further contributions include the latest cutting-edge findings in light scattering made by Klehm et al. [KSE14] and Peters et al. [Pet+16] with very high performance at 1920×1080 pixels.

Figure 3.7. A very realistic raymarched volumetric cumulus cloud [Bou08].

Figure 3.8. Volumetric neural network generated cloud [Kal+17].

Figures 3.7 and 3.8 are volumetric rendering cloud examples. They use thousand of rays to evaluate noise hypertexture samples and calculate an accurate lighting system. In spite of the high realism, the real-time performance is not efficient enough in some contexts, on the other hand. As an example, the work of [Kal+17] uses complex neural networks to render a frame every 10 minutes.

3.3 Main thesis contributions

The main contributions of this thesis to the state of the art are discussed in the following lines as an attempt to illustrate the author's efforts in this computer graphics research field. For a better clarification of this section, contributions are detailed separately for each rendering method.

3.3.1 To texturized primitives

The main contributions of this thesis to the texturized primitive method is the availability of approaching, manoeuvring around, and traversing the gaseous region, in addition to the capability of cloud entity animation. Similarly, the cloud responds to sunlight depending on its position in real-time. Finally, the effects of perspective are more real than those of skydome or skybox projected clouds.

3.3.2 To particle systems

The author of this thesis decided to use pseudospheroids as the basic primitive instead of particles to allow the animation of the cloud with reduced system overhead. The realism of each pseudospheroid is better than that achieved using a collection of texturized particles. This method is very convenient to produce time-evolving cumuliform clouds, as will be explained in Chapter 7.

3.3.3 To geometry distortion

The improvements achieved in this thesis in the area of distorted megaparticles include the possibility to use arbitrary-shaped clouds other than cumulus, for example, thin and subtle clouds (cirrus) or L-system jet streams. Another advantage is the availability to use better lighting models based on multiple scattering and the animation effect caused by wind advection at the frayed edges.

3.3.4 To volumetric rendering

The contributions of this thesis to this emerging and "de facto" method are the improvement of the performance of existing algorithms and the development of new and efficient algorithms and methods to reduce the physics workload of many slow hyperrealistic implementations without loss of quality. The mentioned methods are well suited for animation and real-time rendering on standard computers by using lightweight and ontogenetics-based fluid dynamics algorithms.

3.4 Pros and cons of each method

	Texturized primitives	Particle systems	Geometry distortion	Volumetric rendering
Procedural	✓	✗	✗	✓
Animation	✗	✓	✓	✓
Traverse	✗	✓	✗	✓
Interactive	✓	✓	✓	✓
Manoeuvring around	✗	✓	✓	✓
All kinds of clouds	✓	✗	✗	✓
Realism	✓	✗	✓	✓
Performance	✓	✓	✓	✗
State of the art	✓	✗	✗	✓

Table 3.1. Features per method.

	Mukina et al. [MB15]	Harris [Har03]	Kniss et al.[Kni+02]	Kallweit et al. [Kal+17]	de Parga [JG18]
Procedural	✓	✗	✗	✓	✓
Animation	✗	✓	✓	✗	✓
Traverse	✗	✓	✗	✗	✓
Interactive	✓	✓	✓	✓	✓
Manoeuvring around	✗	✓	✓	✓	✓
All kinds of clouds	✓	✗	✗	✗	✓
Realism	✓	✗	✓	✓	✓
Performance	✓	✓	✓	✗	✓
State of the art	✓	✗	✗	✓	✓

Table 3.2. Features per author.

As seen in Table 3.1 the volumetric rendering method is the most advantageous method in the list of state-of-the-art methods in the computer cloud generation field, fulfilling all the requirements. The rest of the methods lack some important features such as realism or animation effects. However, some volumetric rendering models still need performance improvement. Table 3.2 provides a comparison of features per author. As emphasized in the work of Mukina et al. [MB15], that work meets the basic requirements for the architectural industry but lacks three-dimensional effects. That work falls in the texturized primitives category. In addition, the research by [Har03] is an example of the particle-system-based method, so it represents the opposite end of the preceding cited work. The investigation of [Kni+02] is a case of geometry distorted clouds, whereas the neural network model of [Kal+17] from Disney Research is an example of very accurate photorealistic cumulus clouds; however, it lacks real-time capability. Finally, the research by [JG18] is a demonstration of a new contribution to the state of the art by implementing efficient algorithms for real-time GPU volumetric cloud rendering.

3.5 Summary

This chapter aims to illustrate the main cloud rendering methods from the emergence of computer graphics to the latest trends in this research field. It seeks to present the strengths and weaknesses of each rendering model as an attempt to find solutions that resolve the shortcomings of each system. As a counterpart, the author delineates and demonstrates his contributions to the state of the art in relation to other authors' works and by presenting novel features of his work.

CHAPTER 4

Cloud rendering

THis chapter introduces the basic methods for cloud rendering from the procedural texture generation to the scene delimitation. For this reason, in Section 4.1 is illustrated the three-dimensional structure to generate random noise in the simplest version. In Section 4.2 a complete description of the hypertexture generation and demonstration is provided, along with a complete plot diagrams that compare the produced noise with Gaussian distribution. The cloud rendering methods will be explained in Section 4.3 through the definition of the pseudosphere concept, the development of the basic rendering algorithm layout and the cloud generation detail when approaching to the observer. Finally, in Section 4.4, a brief outline of this thesis approach for locating cloud primitives is given.

4.1 Water vapour emulation

Fluid and asymmetrical cloud shapes were a serious challenge during this research. An efficient 3D data structure that can hold density information and respond to light and wind advection is very convenient for a real-time cloud rendering. However, raytracing of three-dimensional points in real-time requires hardware support by a GPU shader technology that parallelizes the raymarching of a discrete frame buffer pixels by using multiple processing elements.

To create an efficient vapour density simulation, a three-dimensional cube of $64 \times 64 \times 64$ single-precision floats is filled with uniform random noise pre-calculated in the host before passing it to the GPU where it is used to generate fractal Brownian motion (fBm) noise, as shown in Figure 4.1:

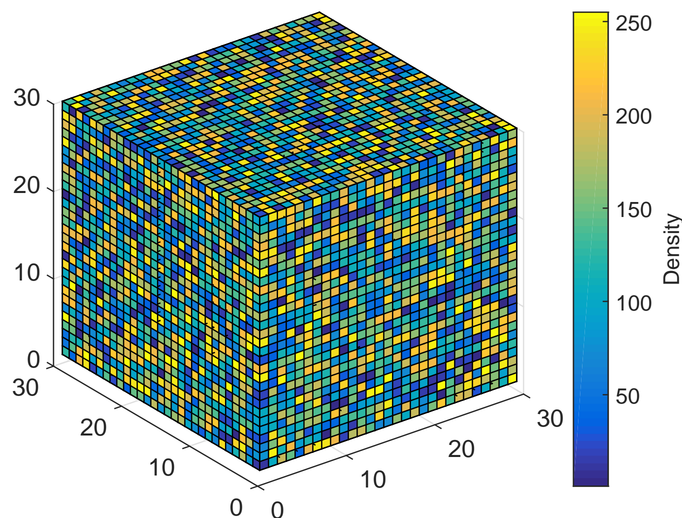


Figure 4.1. The uniform noise in the colour scale plot shows the irregular density of water droplets in a cloud hypertexture. In the MATLAB fourth dimension plot it is possible to observe the nature of atmospheric vapour in the colour values in the cube plot.

4.2 Hypertexture generation

4.2.1 Introduction

The uniform noise is transformed into a Gaussian-like noise using the fBm described in the Iñigo Quilez' homepage [Qui18].

Basically an *fBm* can be implemented as a fractal sum of *Perlin* noise functions [Per85]. The *Perlin* noise is basically a pseudorandom mapping of R^d into R with an integer d which can be arbitrarily large but which is usually 2, 3, or 4. It is used as procedural texture in Computer Graphics to emulate controllable pseudorandom appearance with the aim of generating a wide variety of object surfaces, fire, smoke and clouds. This noise usually imitates the textures in nature due to its stochastic properties. The implementation implies three steps : (1) n-dimensional grid definition with random gradient vectors, (2) computation of the dot product between the gradient vectors for distance calculation and (3) the interpolation of the mentioned dot products. The algorithm cost is $O(2^n)$ where n is the number of dimensions.

4.2.2 The *fBm* noise

As cited in [Die04], the *fBm* noise is defined by Mandelbrot and van Ness [MV68] by its stochastic representation 4.1.

$$B_{H(t)} = \frac{1}{\Gamma\left(H+\frac{1}{2}\right)} \left(\int_{-\infty}^{0} \left[(t-s)^{H-\frac{1}{2}} - (-s)^{H-\frac{1}{2}} \right] dB(s) + \int_{0}^{t} (t-s)^{H-\frac{1}{2}} dB(s) \right) \tag{4.1}$$

where Γ represents the gamma function and $0 < \mathrm{H} < 1$ is called the Hurst parameter 4.2.

$$\Gamma(\alpha) = \int_0^{\infty} x^{\alpha-1} e^{-x} dx \tag{4.2}$$

The integrator B is a stochastic process, ordinary Brownian motion, where B is recovered by taking $H = 1/2$ in Equation 4.1. This integral can be seen as a Lebesgue-Stieltjes integral. It is called the stochastic integral with respect to Brownian motion.

$$\int_a^b \phi(s) dB(s) \tag{4.3}$$

Equation 4.3 has a natural definition when the integrand ϕ is a simple function, which means that there exists an integer $\gamma > 0$ and a strictly increasing sequence of real numbers $(t_j)_{j=0}^{\gamma}$ with $t_0 = a$ and $t_\gamma = b$, in addition to a sequence of real numbers $(\phi_j)_{j=0}^{\gamma-1}$ such that $\phi(s) = \phi_j$ for $s \in (t_j, t_{j+1}]$. Thus, the stochastic integral is defined in Equation 4.4:

$$\int_a^b \phi(s) dB(s) = \sum_{j=0}^{\gamma-1} \phi_j \left(B(t_{j+1}) - B(t_j) \right) \tag{4.4}$$

A sequence of Lebesgue square integral functions (ψ_n) is said to converge in the L^2-norm to the square integral function ψ if Equation 4.5 applies:

$$\lim_{n\to\infty} \int (\psi(s) - \psi(s))^2 ds = 0 \tag{4.5}$$

assuming that ψ is a simple function for every n. A possible definition would be $\lim_{n\to\infty} \int \psi_n(s) dB(s)$. Since $\int_a^b \psi(s) dB(s)$ has a finite second moment, there is a square integrable random variable Z on the probability space of $B(\Omega, F, P)$ with the property stated in Equation 4.6:

$$\lim_{n\to\infty} \mathbb{E}\left[\left(Z - \int_a^b \psi_n(s) dB(s) \right)^2 \right] = 0 \tag{4.6}$$

where $\mathbb{E}$ denotes the expectation operator with respect to P. If ψ is square integrable, this random variable is unique in the sense that another random variable that satisfies Equation 4.6 is almost surely equal to Z. Therefore, Z is taken as the definition of the stochastic integral $\int_a^b \psi(s) dB(s)$. The second moments of Equation 4.1 can be computed by a standard formula, which leads to the fact that the variance of $B_H(t)$ is $V_H t^{2H}$ for some constant V_H. Given $B_H = \{B_H(t) : 0 \le t < \infty\}$ with 0 < H < 1, the following properties are characterized:

- $B_H(t)$ has stationary increments;
- $B_H(0) = 0$, and $\mathbb{E}B_H(t) = 0$ for $t \ge 0$;
- $\mathbb{E}B_H^2(t) = t^{2H}$ for $t \ge 0$;
- $B_H(t)$ has a Gaussian distribution for t > 0;

The covariance function for the three first properties is given in Equation 4.7:

$$\rho(s,t) = \mathbb{E}B_H(s)B_H(t) = \frac{1}{2}\{t^{2H} + s^{2H} - (t-s)^{2H}\} \quad (4.7)$$

for $0 < s \leq t$. For Gaussian processes, the mean and covariance structure determine the finite-dimensional distribution uniquely. Therefore, we conclude from Equation 4.7 that $\{B_H(at) : 0 \leq t < \infty\}$ and $\{a^H B_H(t) : 0 \leq t < \infty\}$ have the same finite-dimensional distributions: the fractional Brownian motion with the Hurst parameter H is self-similar with the Hurst parameter H. In fact, fractional Brownian motion is the only Gaussian process with stationary increments that is self-similar. The *fBm* noise also has applications in communications, finance, physics and bioengineering.

4.2.3 More about the *fBm* noise

The implementations of this thesis makes intensive use of the *fBm* noise as a summation of weighting uniform noise. Thus, let w be the octave scale factor, s the noise sampling factor and i the octave index, the fBm equation is defined as:

$$fbm(x,y,z) = \sum_{i=1}^{n} w^i \cdot perlin(s^i x, s^i y, s^i z) \quad (4.8)$$

where $w = 1/2$ and *s* is 2.

Each iteration is called an octave. In this case, the Algorithm 4.1 uses five octaves with uniform noise instead of *Perlin* noise. When the number of octaves is increased above five, the algorithm does not produce better vapour deformation shape, but decreases the overall frame performance. To analyse performance, an unrolled version of the previous summation was implemented in the thesis fragment shader code.

As commented in the introduction of Section 4.2, the uniform noise used for the *fBm* generation has the aspect of Figure 4.2 on the left, while the proper *fBm* has the look plotted in Figure 4.2 on the right. This soft appearance will be useful for cloud texture rendering. The probability distribution of these two kind of noises can be appreciated in the histogram of Figure 4.3.

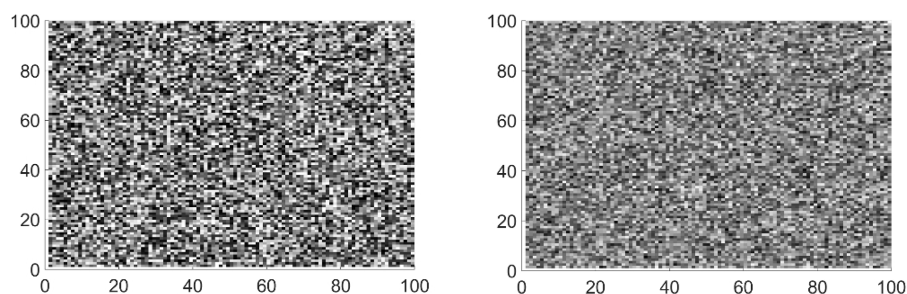


Figure 4.2. Bi-dimensional representation of uniform noise (**left**) and fBm noise (**right**). It can be observed that the base noise on the left is sharper while the fBm is softer due to the octave accumulation.

Other authors (see for example [HSA05]) also make use of fractional octave division in their research with respect to the 2D flat animation of clouds.

The *fBm* noise is calculated for each unit of the frame buffer along the tracing of the ray to produce cloud density. Alternatively, the noise may be pre-calculated in the host application before passing it to the shader to achieve higher frames-per-second. If the basic pre-calculated 3D noise is used for all the scene, a simple raymarching algorithm may be used, as explained on the next page.

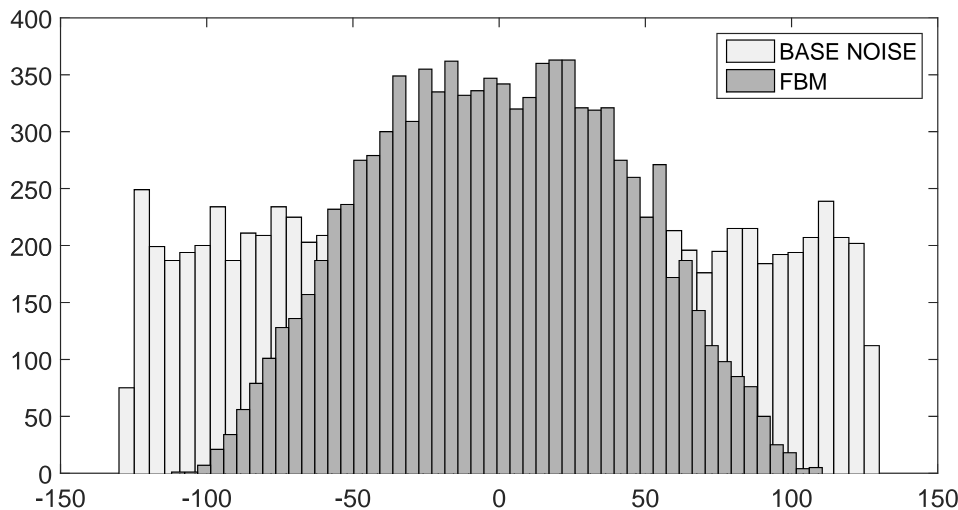


Figure 4.3. Histogram of the uniform noise (light grey) and fBm noise (dark grey). The fBm noise has a Gaussian-like distribution.

4.3 Cloud tracing

4.3.1 Pseudospheroids

For every pixel in the frame buffer, the ray is traced form back to front to calculate the noise with the condition that the density value is smaller than γ. This variable is modulated as a function of the position inside the pseudospheroid with a random component to produce the effect of cloud surface as shown in Equation 4.9:

$$\gamma \leftarrow e^{\left[\frac{-\|rayPos - s\vec{phereCenter}\|}{radius \times \left(\left(1-\kappa\right)+2\kappa fbm(x,y,z)\right)}\right]} \tag{4.9}$$

The appearance of the pseudospheroids can be appreciated in Figure 4.4, where R is the radius of the base sphere and γ is the value provided by the function in Equation 4.9. The pseudosphere objective is to emulate a more irregular and realistic primitive form for cumulus clouds.

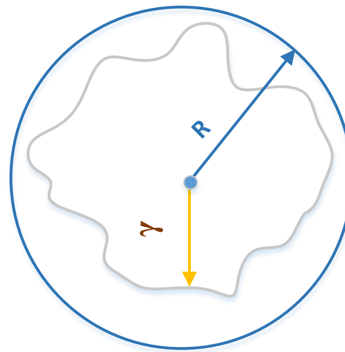


Figure 4.4. Pseudospheroid shape in 2D satisfying Equation 4.9.

The Equation 4.9 filters the unit volume densities and generates a pseudospheroid by scaling the nominal sphere radius by $\pm$ k, where $rayPos$ is the Euclidean straight line point of the ray and $sphereCenter$ is the center of the sphere in R^3. The *fBm* function serves as a normalized random variable. The distance to the center of the sphere is used to modulate the maximum density that water vapour is allowed to have, as shown in Figure 4.5.

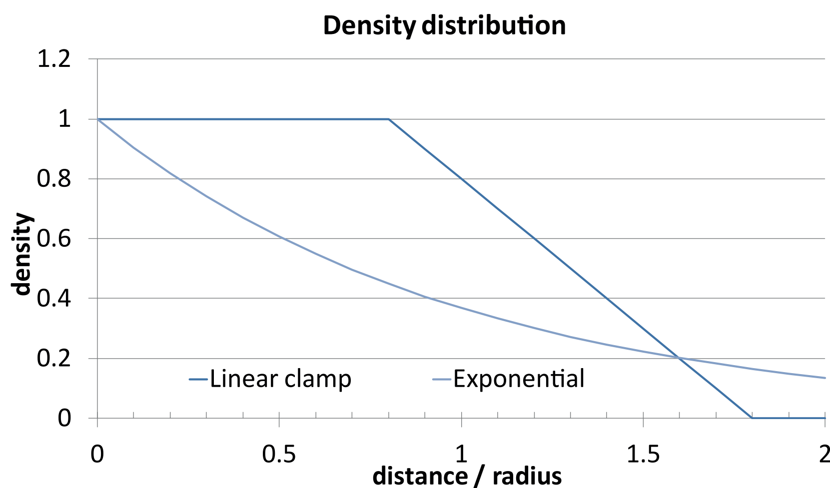


Figure 4.5. Distance/density relationship. The greater the distance between the ray and the sphere center, the lower the density of water vapour. The exponential approach gives natural realism by softening the cloud borders.

4.3.2 Basic cloud rendering algorithm

The rendering of the cloud involves tracing rays through a volume delimited by a set of implicit surfaces, while sampling the noise texture. The sampling of the noise texture is performed only when the ray traverses the volume defined by a set of spheres. The pseudocode scheme in Algorithm 4.1 is an improved version of Apodaca's code [AG00] and basically illustrates the execution over a list of N randomized bounded-surface volumes. The alternative of using spheroids and ellipsoids instead of other surfaces is also cited by Gardner [Gar85] and Elinas and Stürzlinger [ES00]. To discard positions outside the cloud and thus reduce the computational effort, we iterate over the pseudospheroids contained in the corresponding bounding box only. In addition, a short list of pseudospheroids that intersect the ray is created and sorted with the *Insertion Sort algorithm* [Wei95; Knu98]. This collision method requires the computation of the discriminant of the quadratic equation resulting from equaling the implicit formulas of rays and spheres using the reference code based on Shirley's book [SK03].

Basically, the Algorithm 4.1 starts in line 27 with the *rayTrace* function, tracing a ray for each pixel in the frame buffer within an origin and direction. After cloud limits detection in line 28, the algorithm begins to iterate over a set of detected clouds by selecting the spheres that intersect the ray (lines 2 - 8) with a simple Euclidean discriminant (line 7). If this discriminant is positive, the algorithm inserts the sphere's near and far collision limits (λ) in a candidates array (line 12). Retaking the main function, the algorithms continues by sorting the collided spheres in ascending order, from close to far, using the *Insertion Sort* algorithm. After this operation, the algorithm iterates over each sphere (lines 34 - 51) one by one rendering the pseudosphere (line 40 and 41)and lighting the clouds (lines 42 - 46). Once the chain of spheres has been rendered, the resulting pixel is blended with the sky by considering transparency (lines 30 and 52).

```
1  Function getCandidates(rayOrigin, rayDirection→, i)
2      for j ← 0 < number of spheres in boundingbox[i] do
3          temp→ ← rayOrigin − sphereCenter→
4          a ← rayDirection→ · rayDirection→ {Dot products}
5          b ← 2.0 · rayDirection→ · temp→
6          c ← temp→ · temp→ − radius²
7          Δ ← b² − 4ac
8          if Δ > 0 then
9              σ ← √Δ {There is a collision}
10             λ_in ← (−b−σ)/(2.0×a)
11             λ_out ← (−b+σ)/(2.0×a)
12             candidates[k] ← (λ_in, λ_out, j, i)
13             k = k + 1
14         end
15     end
16     return (candidates,k)
17 Function sortCandidates (candidates, n)
18     for k ← 1 < n do
19         aux = candidates[k] {Insertion-sort algorithm}
20         h = k − 1
21         while ((h >= 0)and(aux_λin < candidates[h]_λin)) do
22             candidates[h + 1] = candidates[h]
23             h = h − 1
24         end
25         candidates[h + 1] = aux
26     end
27 Function rayTrace(rayOrigin, rayDirection→)
28     B ← boundingboxDetection(rayOrigin, rayDirection→)
29     τ ← 1
30     R ← (0,0,0,0) {Consider alpha-channel}
31     for each (i in B) do
32         (C,n) ← getCandidates(rayOrigin, rayDirection→, i)
33     end
34     candidates ← sortCandidates(C,n)
35     for  j ← 0 < n do
36         λ ← candidates[j]_λin
37         λ_out ← candidates[j]_λout
38         while λ ≤ λ_out do
39             rayPos ← rayOrigin + λ · rayDirection→
40             ρ ← fbm(rayPos) {Trace pseudospheroid}
41             γ ← e^(−||rayPos − sphereCenter→|| / (r((1 − κ) + 2kρ)))
42             if ρ < γ then
43                 R ← R+ lighting(ρ, rayPos, rayDir→, sunDir→,
44                                 voxelGrid[candidates[j]_i], sunColor)
45                 if τ < 10^−6 then
46                     break for (34:)
47                 end
48             end
49             λ ← λ + A · e^−(||rayOrigin − cloudCenter→|| · δ) {LOD}
50         end
51     end
52     return R {Blend with the sky}
```

Algorithm 4.1: Basic volumetric cloud rendering in GPU.

4.3.3 Level of detail (LOD)

An improved level of detail (LOD) equation controls the increment on the Euclidean straight line [Gil88] according to the distance from the center of the cloud, thus executing longer steps when the cloud is close to the camera and smaller steps when it is far. This method compensates the higher number of pixels receiving a trace from inside the volume with the lower requirements for detail when traversing the cloud.

The aim of this technique is to provide more efficient distance effects for computer graphics applications that do not require high detail when the observer is near or inside the cloud mass. This aspect is mainly useful for computer games and virtual reality educational software where there is no need to analyse the texture details.

$$\lambda = A \cdot e^{-(\| rayOrigin - \vec{cloudCenter} \| \cdot \delta)} \tag{4.10}$$

Equation 4.10 represents the LOD behaviour for the Euclidean straight line increment in relation to the distance between the observer and the centre of the cloud. This equation relates to line 48 of Algorithm 4.1.

The previous equation is obtained if it is considered that $\lambda = A \cdot e^{-distance \cdot \delta}$ and, in the present implementation, λ ranges from 0.1 at infinite distance to 10 when the cloud center is at distance 20 from the camera. Therefore, solving δ in the next equation:

$$\delta = -\frac{ln\left(\frac{\lambda}{A}\right)}{distance} \tag{4.11}$$

And replacing this constant in Equation 4.10, we obtain the plot of Figure 4.6:

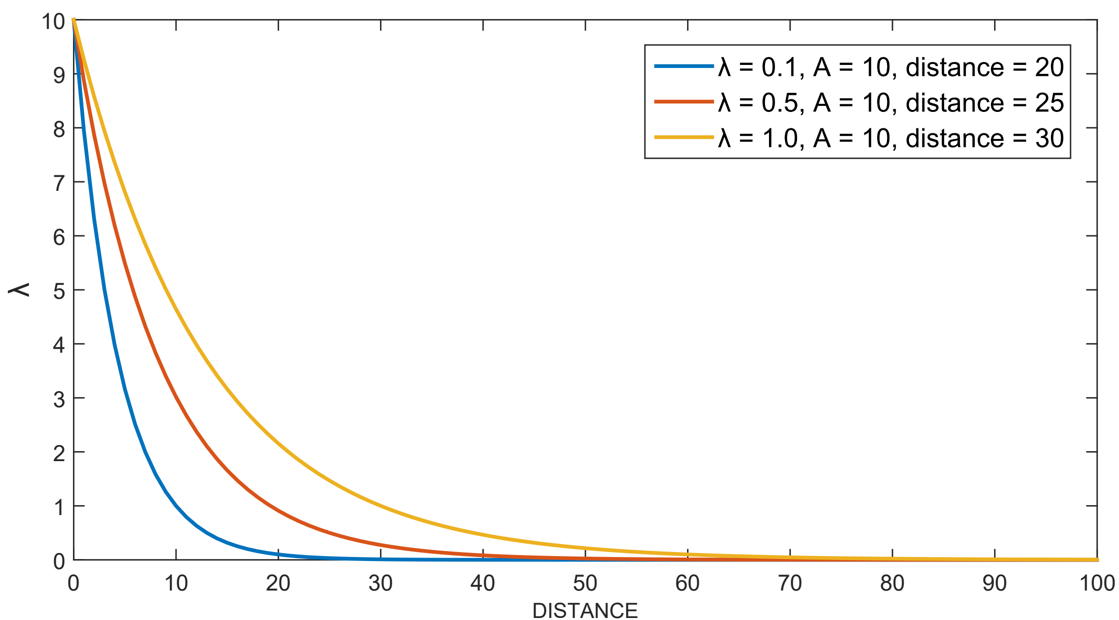


Figure 4.6. Three plots of different coefficients for Equations 4.10 and 4.11. The X axis represents the typical distance of cloud rendering from nearby (0) to the camera being far away (100). The Y axis represents the achieved step size at the specified distance and δ constant.

4.4 Scene delimitation

The technique explained hereby eliminates the need to use complex space-partitioning algorithms like K-D trees, Octrees, or Quadtrees (see Section 2.2) for locating primitives thanks to the fact that each cloud only needs a small number of spheres to create a cumulus in this proposed model. In addition, the use of bounding boxes to delimit the volume of each cloud also improves performance. The proposed system uses Smits' algorithm [Smi02] and the improvement in [Wil+05] to limit raymarching to the volume inside cloud boundaries, as shown in Figure 4.7.

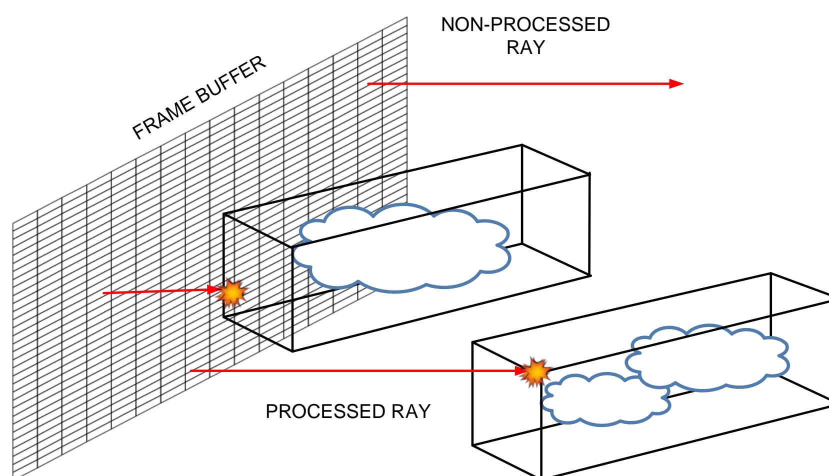


Figure 4.7. Bounding box delimitation of clouds. The Smits' algorithm is useful for optimizing raymarching by calculating the segment of the Euclidean straight line along which rendering must be performed according to the λ value.

This method increases the FPS rate by discarding rays that never hit the boundaries of the rectangular prism containing the cloud in the scene, and avoiding iterations over spheres outside the camera view frustum. As an additional advantage, the bounding box model allows for assigning a separate voxel grid to each cloud, which facilitates the calculation and storage of illumination data. Each spheroid in the candidates' lists has a pointer to its corresponding voxel grid, as seen in Algorithm 4.1. The improved Smits' algorithm is implemented in the GPU code in a fragment shader with a very reduced linear cost.

Since each bounding box contains an array of pseudospheres with R^3 coordinates and a radius in the form of GLSL **vec4** style, the shader application makes use of another array of pointers to the lower limit and upper limit of the corresponding set of spheres in the first array, as Figure 4.8 illustrates. The second array of pointers is useful for redirecting the collision detection function to the specific clouds under the raymarching process.

Because of the fact that the cloud rendering model of this thesis needs only approximately

30 spheres to generate a cumulus, the resulting bounding box iteration is very efficient, as will be seen in Chapter 6.

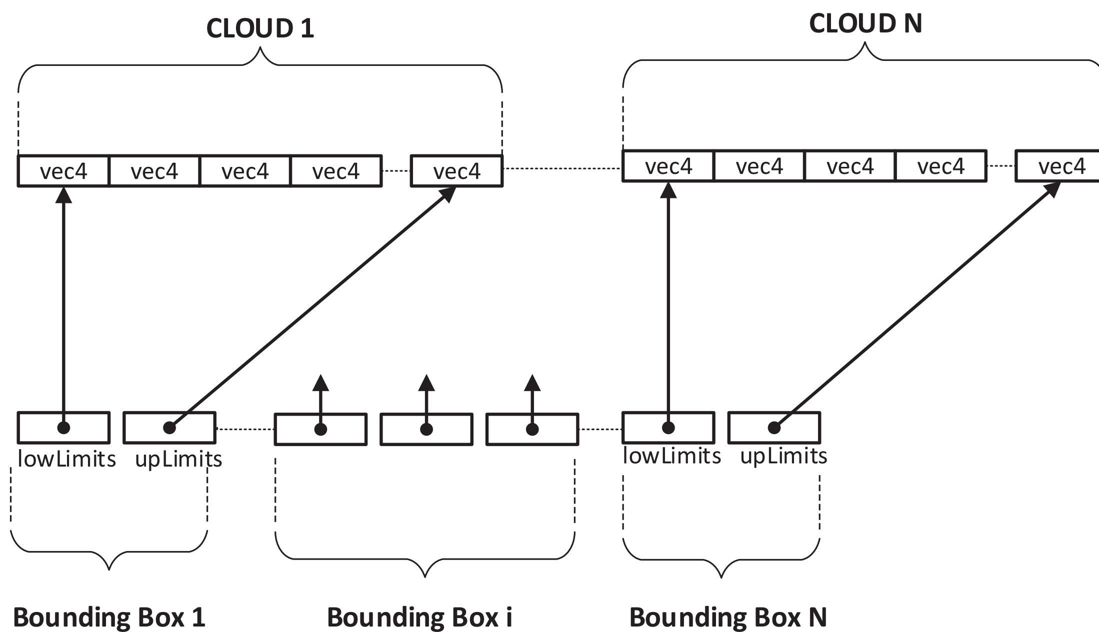


Figure 4.8. Each cloud has a set of pseudospheres distributed in a 1D array (above). When the ray collides with a bounding box, their spheres are processed as a result of the lower and upper limit array of the pointers (below) in the GLSL fragment shader.

4.5 Summary

This chapter illustrates the basics of cloud rendering methods, including water vapour generation, along with the mathematical demonstration of the *fBm* noise used to evaluate the hypertexture of the clouds. A complete illustration of the basic cloud rendering algorithm in GPU with the use of pseudospheroids is explained as well. The chapter ends by explaining the present thesis' efficient method to delimit and locate cloud primitives in the GPU shader code.

CHAPTER 5

Lighting and shading

THE present chapter aims to introduce the main concepts and prior literature references used for creating the thesis lighting and shading model. Thus, in Section 5.1, a brief outline of the main referenced authors and main contributions to the state of the art are introduced. Next, in Section 5.2 a complete explanation of the lighting and shading model is developed. The intricacies of the no-duplicate-tracing (NDT) algorithm are exposed in Section 5.3 along with its flow chart, and the details of the parallel shading precalculation is introduced in Section 5.4. Finally, a set of images illustrates the achieved lighting objectives in Section 5.5.

5.1 Introduction and contributions

To optimize the performance, this thesis uses a two-pass drawing algorithm. The first pass computes the illumination inside the cloud and is executed only when the location of the light source changes. The second pass renders the projection of the cloud on the frame buffer. The calculation of transmittance and scattering are inspired by the principles exposed by Max [Max95] with code optimizations to improve performance with the application of approximations by Harris [HL01] and Tessendorf [Tes16] in the pseudospheroid-based volumetric rendering of this theis. The main differences with these works are the application of the ray-marching technique over a set of pseudospheroids/ellipsoids instead of a particle system, the use of a pre-calculated voxel grid with the NDT algorithm, and the combination of simplifications methods to generate a new model of code optimization as seen in Algorithm 5.1. The implemented model makes use of the *voxel* technique, which divides the space in a 3D grid with regular cells. These cells may store information light transmittance, density and other physical parameters. The voxel technique is currently used in CT and MRI systems in medicine. Essentially, the main contributions to the previously cited authors' work are:

- The combination of three different techniques in a pseudospheroid based volumetric rendering.
- The use of the no-duplicate-tracing (NDT) algorithm for transmittance precomputation.
- The simplification of the referenced algorithms to achieve high performance in standard computers.

5.2 Lighting and shading model

Algorithm 5.1 illustrates the required steps for calculating the lighting and shading in the GPU shader. The lighting function is called in Algorithm 4.1, line 42. The performed calculations are widely detailed in the following paragraphs.

```
1  Function lighting(ρ,rayPos,rayDir⃗,sunDir⃗,gridIndex,sunColor)
2      Δτ ← e^(−φρ)
3      {Calculate Voxel}
4      voxelIndex ← (rayPos−gridMin)/(gridMax−gridMin)
5      {Precomputed-light retrieval}
6      pL ← texture(gridIndex,voxelIndex)
7      {Absorpted-light calculation}
8      aL ← sunColor ∘ pL
9      {Scattered-light calculation}
10     sL ← sunColor · phase(g,rayDir⃗,sunDir⃗) · pL
11     {Total-light calculation}
12     tL ← aL · κ1 + sL · κ2
13     {Calculate cloud colour}
14     colour ← (1 − Δτ) · tL · τ
15     τ ← τΔτ
16     return (colour,1 − τ)
```

Algorithm 5.1: Lighting and shading.

The mentioned algorithms are illustrated in Figure 5.1 where the green ray represents the illumination pass pre-computed in the CPU and the red ray represent the rendering pass performed in the GPU. The illumination pass traces a ray *l* from each voxel *v(l)* inside the volume of the cloud towards the light source *I*, covering a distance *D*. The voxel stores the amount of light received by transmission and scattering in that point of the cloud. This part is pre-calculated in the CPU. The render pass traces a ray *s* that traverses the entire cloud from 0 to *P* to calculate the light projected on the frame buffer, taking into account the contribution of emission and reflection. This part is executed in real-time in the GPU.

5.2.1 Transmittance

This calculation is performed one time for the pre-computed light and in each pass for the rendering. The attenuation of light inside the cloud is governed by a property called transmittance that represents the amount of light passing through a given section of the volume. All the light inside the volume of the cloud, whether it is absorbed or scattered, is affected by transmittance. The amount of light passing along a ray from point s to point s' is expressed as:

$$\tau(s) = \rho(s) \cdot A \tag{5.1}$$

$$T(s,s') = e^{\int_s^{s'} \tau(t)dt} \tag{5.2}$$

where $\tau(s)$ is the extinction coefficient that represents the amount of area that occludes light in a plane perpendicular to the ray, $\rho(s)$ is the particle density of water vapour at location s, and A is the area occluded by the particle projected on the plane perpendicular to the ray. For the forthcoming expressions we will use the simplified notation $T(s,s')$.

5.2.2 Illuminations pass (CPU side)

The first pass computes the illumination inside the cloud assuming that the light source is static, and therefore the calculations are performed only when the light source changes location or intensity. The illumination inside the cloud is stored as a grid of voxels using floating point scalars. The light collected in a voxel is the sum of the light transmitted from the light source across the volume plus the light scattered from all the preceding points [1] along the ray between the voxel and the light source. This second term involves a summation of multiple light contributions with their corresponding transmittance. Illumination of a voxel is computed using a back-to-front raymarching algorithm with the NDT method.

Let $L(v)$ be the light collected at point v inside the cloud, I_0 the intensity of the light source, and D be the depth of v from the surface of the cloud. The expression that represents the light incident on point v is:

$$L(v) = \underbrace{I_0 \cdot T(0,D)}_{\text{absorption}} + \underbrace{\int_0^D C(l) \cdot T(0,l)dl}_{\text{scattering}} \tag{5.3}$$

The first term in Equation 5.3 is the light propagated across the volume of the cloud to point v that is affected by the total transmittance form the surface to the point. The second term is the light scattered in the forward direction along the ray *(l)* from the surface of the cloud to the destination point. This term is the summation of the light emitted by scattering from each intermediate point along the ray, attenuated by the corresponding transmittance of the segment.

[1] This thesis applies a simplified multiple forward scattering technique.

The term $C(l)$ is the approximation of the forward light-scattering density according to [HL01]. The mentioned work reports that 50% of the light scatters in the forward direction of the ray over a small solid angle γ that is of about 0.0001 steradians. The remaining 50% is ignored because it requires a large amount of computation.

$$C(l) = \frac{1}{2}\frac{\gamma}{4\pi} \quad (5.4)$$

Equation 5.3 is converted to discrete form to be calculated using a raymarching algorithm with step size Δl. T_i is the transmittance between the target point and point i along the ray:

$$L(\vec{v}) = \sum_{i=0}^{D/\Delta l} \frac{1}{2}\frac{\gamma}{4\pi} \cdot T_{t-1} + I_0 T_D \quad (5.5)$$

5.2.3 Render pass (GPU side)

The second pass renders the projection of the cloud on the frame buffer using a raymarching algorithm that takes into account reflection and scattering.

The light reflected at point s inside the cloud depends on the light present at this point *L(s)* and the area occluded by particles $\tau(s)$. The reflected light is further affected by the transmittance form point s to the surface of the cloud.

The light scattered at point s is affected by the same properties as the reflected light but also the phase function that approximates the refraction of light depending on the angle between the light source and the observer. The amount of light projected on point $\vec{x}$ of the frame-buffer is expressed as:

$$I(\vec{x}) = \int_0^P \left[\underbrace{L(s)\tau(s)\kappa_1}_{\text{reflection}} + \underbrace{L(s)\tau(s)P(\vec{n}_s,\vec{n}_l)\kappa_2}_{\text{scattering}} \right] \cdot T(0,s)ds \quad (5.6)$$

The first term in Equation 5.6 accounts for reflected light, and the second term accounts for scattered light. $P(\vec{n}_s,\vec{n}_l)$ is the phase function representing the amount of light from direction $\vec{n}_l$ as refracted in direction $\vec{n}_s$. Both terms are affected by the transmittance T(0,S) from the surface to point s. The κ_1 and κ_2 coefficients are weighting constants.

$$I(\vec{x}) = \sum_{i=0}^{P/\Delta s} \int_0^{\Delta s} L(i\Delta s + t)\left[\kappa_1 + P(\vec{n}_s,\vec{n}_l)\kappa_2\right] \cdot \tau(i\Delta s + t) \cdot T(0,i\Delta s + t)dt \quad (5.7)$$

To compute the emitted light using a raymarching algorithm with step size Δs, this thesis used the approximation proposed by [Tes16] that converts the continuous integral into a summation of segment integrals. T_i represents the transmittance from the surface to point i. The phase term is considered constant along the segment integral and can be taken out.

$$I(\vec{x}) = \sum_{i=0}^{P/\Delta s} \left[T_i\kappa_1 + T_i P(\vec{n}_s,\vec{n}_l)\kappa_2\right] \int_0^{\Delta s} L(i\Delta s + t)\tau(i\Delta s + t)dt \quad (5.8)$$

The differential transmittance ΔT_i along segment Δs is denoted as:

$$\Delta T_i = e^{-\int_0^{\Delta s} \tau(\vec{x}+s\vec{n})ds} \tag{5.9}$$

The exponential function e can be approximated using the first two terms of its McLaurin series expansion as:

$$\Delta T_i = 1 - \int_0^{\Delta s} \tau(\vec{x}+s\vec{n}_s)ds \tag{5.10}$$

This allows simplifying the segment integral in 5.8 as $1-\Delta i$. We also assume that $L(i\Delta s+t)$ is constant inside the segment integral.

$$I(\vec{x}) = \sum_{i=0}^{P/\Delta s} [T_i\kappa_1 + T_iP(\vec{n}_s,\vec{n}_l)\kappa_2]\, L_i \cdot (1-\Delta T_i) \tag{5.11}$$

The proposed implementation made uses the *Henyey-Greenstein* phase function to approximate the refraction of light in water droplets:

$$P(\vec{n}_s,\vec{n}_l) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g\vec{n}_s\cdot\vec{n}_l)^{3/2}} \tag{5.12}$$

where $\theta(\vec{n}_s \cdot \vec{n}_l)$ is the scattering angle and the parameter g is conveniently equal to the asymmetry factor. To improve performance, the implementation passes the voxel grid lighting in a 3D texture to the GPU. This allows using hardware accelerated trilinear interpolation to obtain the lighting values along the marching ray.

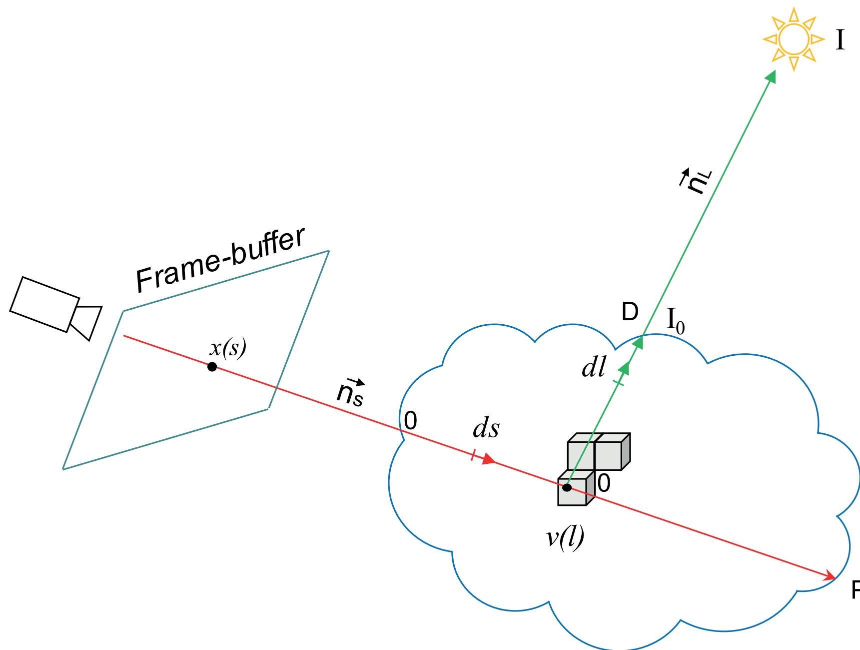


Figure 5.1. A two-pass algorithm for lighting along ray *l* and rendering along ray s .

5.3 The No Duplicate Tracing Algorithm (NDT)

The purpose of this algorithm is to avoid duplicate or void tracing when spheres or ellipsoids overlap or have gaps. This simple method is implemented in a few lines of CPU code to pre-compute the transmittance from the light source to the target voxel, reducing calculations while preserving accurate cloud rendering, as shown in Algorithm 5.2. It applies a Greedy heuristic [BB96] to obtain the optimum set. The image and flow chart in Figures 5.2 and 5.3 respectively explain the possible cases that may arise during the execution of a raymarching.

1 $candidates \leftarrow sortCandidates(C,n)$
2 $\lambda \leftarrow candidates[0]_{\lambda_{in}}$
3 **for** $j \leftarrow 0 < n$ **do**
4   **if** $\lambda > candidates[j]_{\lambda_{out}}$ **then**
5   **continue**
6   **end**
7   **else if** $\lambda < candidates[j]_{\lambda_{in}}$ **then**
8         $\lambda = candidates[j]_{\lambda_{in}}$
9   **end**
10   **while** $\lambda \leq \lambda_{out} = candidates[j]_{\lambda_{out}}$ **do**
11   {Compute transmittance}
12         $\lambda \leftarrow \lambda + \Delta$
13   **end**
14 **end**

Algorithm 5.2: No-Duplicate-Tracing.

This algorithm was tested in the render phase of the GPU side to optimize the performance; however, the control logic of the code prevents a correct pseudosphere illustration, causing rendering errors or aberrations in the overlapped part of the spheres. Despite this unlucky pitfall, the NDT algorithm duplicates the precalculation phase performance in both the CPU and the CUDA implementations.

Others research regarding bounding volumes overlapping includes the work of Lipuš and Guid [LG05].

Algorithm 5.2 has been empirically checked with a set of test cases from basic to complex scenarios in MATLAB with very good results. Currently, there is no formal math demonstration of the algorithm accuracy.

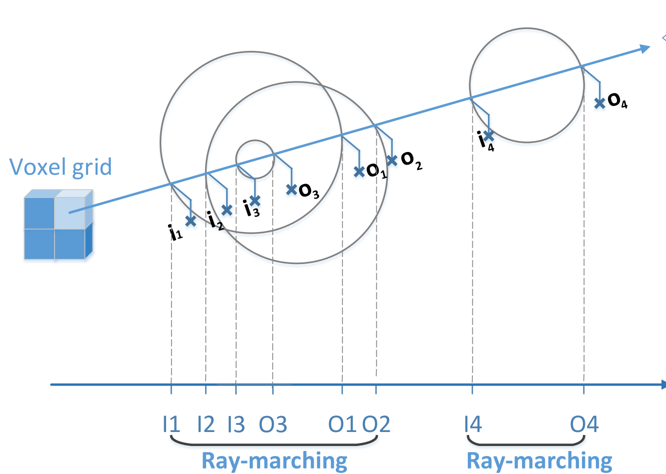


Figure 5.2. A basic model that illustrates the zones to sweep. In this case, only I1 to O2 and I4 to O4 are traced following the ray.

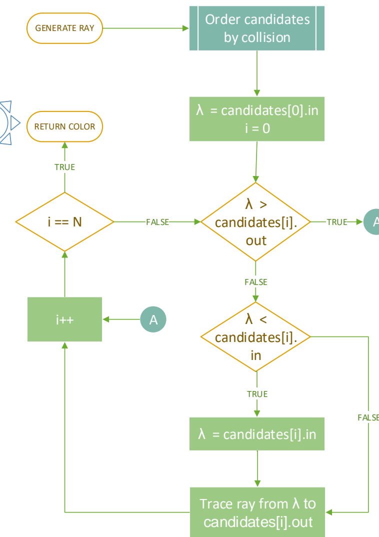


Figure 5.3. A flow chart illustrating the *no duplicate tracing algorithm* (lines 1–14 in Algorithm 5.2

5.4 Precomputed light parallelization

As mentioned in the introductory Chapter 1, the precomputed light calculation has been parallelized with CUDA technology using a different kernel to process each ray from the cloud to the light source and introducing the NDT algorithm in the parallel code.

The CUDA implementation distributes threads (T_x, T_y, T_z) according to the GPU *maxThreadsPerBlock* device capability. Since this variable is usually 1024 in the studied architectures, the multiplication is $T_x \times T_y \times T_z = 1024$. Thus, the distributions of threads in each dimension is $T_x = 16$, $T_y = 8$, and $T_z = 8$ in this case.

Then, it is necessary to calculate the block dimensions with the following equation:

$$Block_{x,y,z} = \left\lceil \frac{Dim_{x,y,z} + 2}{T_{x,y,z}} \right\rceil \tag{5.13}$$

where $Dim_{x,y,z}$ represents the dimensions of the voxel grid.

Therefore, Algorithm 5.2 is processed in each kernel call serially:

```
dim3 blocks(Block_x, Block_y, Block_z)
dim3 threads(T_x, T_y, T_z)
kernel<<<blocks,threads>>>(params...)
```

Once inside the parallel kernel, it is possible to access the grid cells using the following offset:

$$index_{i,j,k} = blockIdx_{x,y,z} \times blockDim_{x,y,z} + threadIdx_{x,y,z}$$

where $blockIdx$ is a CUDA keyword referencing the current block index, whereas $blockDim$ is also a CUDA keyword indicating each block dimension, and $threadIdx$ is another keyword with the current thread index.

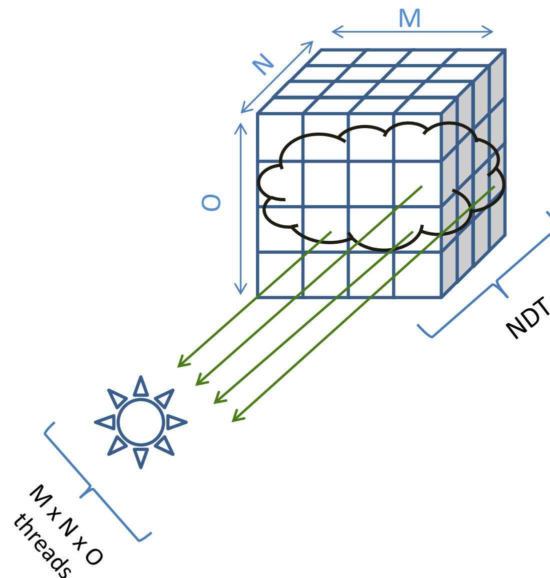


Figure 5.4. The 3D grid with $M \times N \times O$ voxels with a parallel trace of the shadow including the NDT algorithm.

The basic idea behind the parallel implementation is tracing a parallel ray from each voxel in the grid as illustrated in Figure 5.4. When each thread processes a ray in the light source direction, the equations explained in Section 5.2.2 are applied in parallel along with the NDT algorithm.

The CUDA implementation of the lighting precomputing phase reaches a remarkable high performance as demonstrated in the benchmarks section of Chapter 8.

5.5 Lighting samples

The daylight landscape of Figure 5.5 is an example of cloud shading using the NDT algorithm. The pseudospheroid composition can be appreciated.

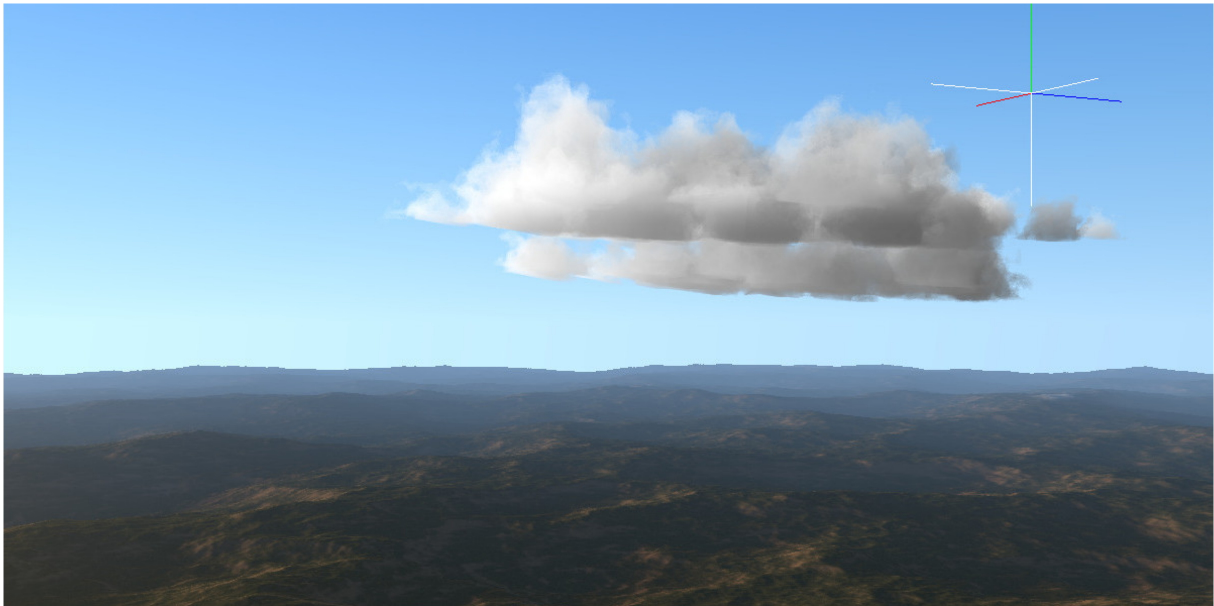


Figure 5.5. An example of shading with the sunlight coming from the left.

The multiple forward scattering can be observed in Figure 5.6 in a daylight scene with the sunlight coming from the front. The silver lining effect is also relevant in this figure.

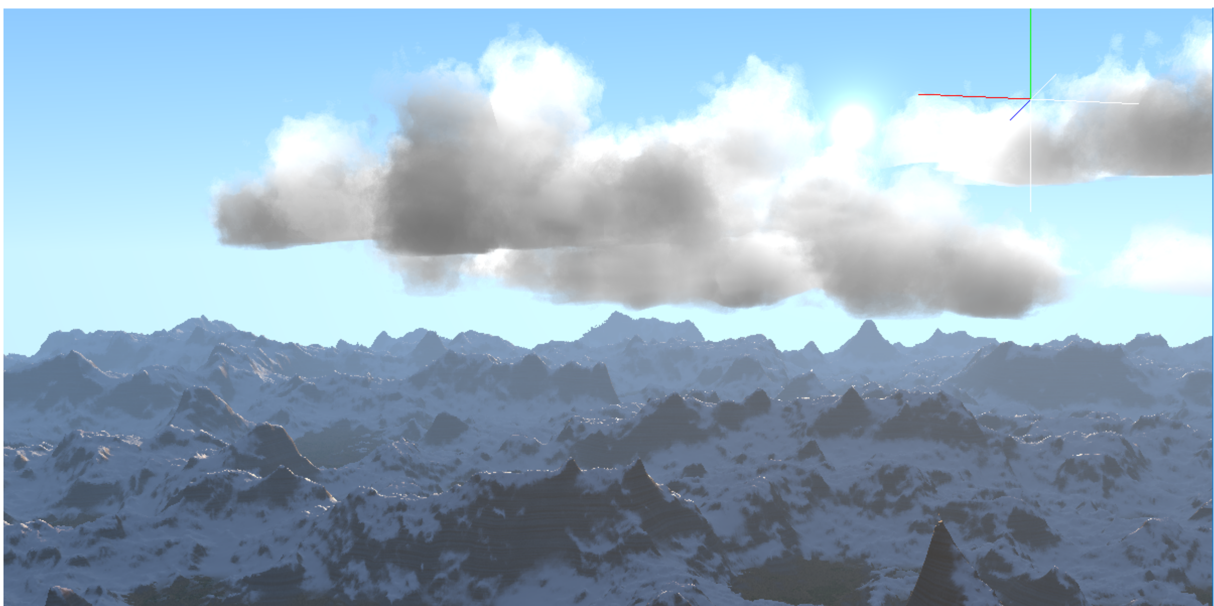


Figure 5.6. A cumulus formation over snowed mountains.

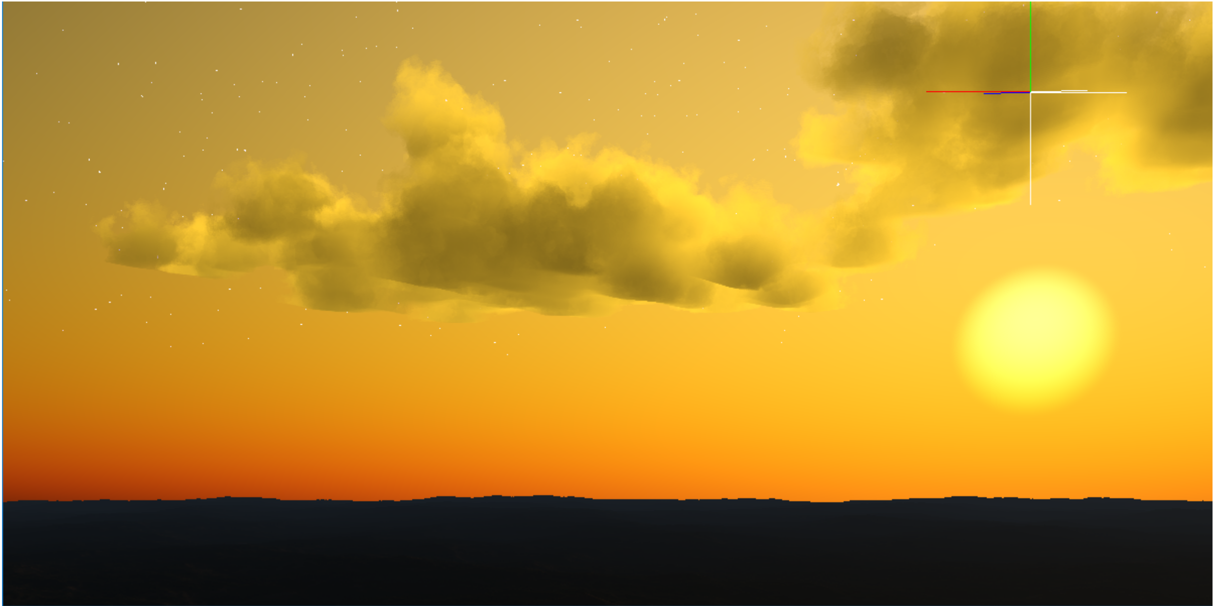


Figure 5.7. A sunset landscape.

A sunset with two cumulus clouds is generated in Figure 5.7. The adjustment of the phase and light intensity and colour allows this kind of scenes. The multiple forward scattering and silver lining is also represented.

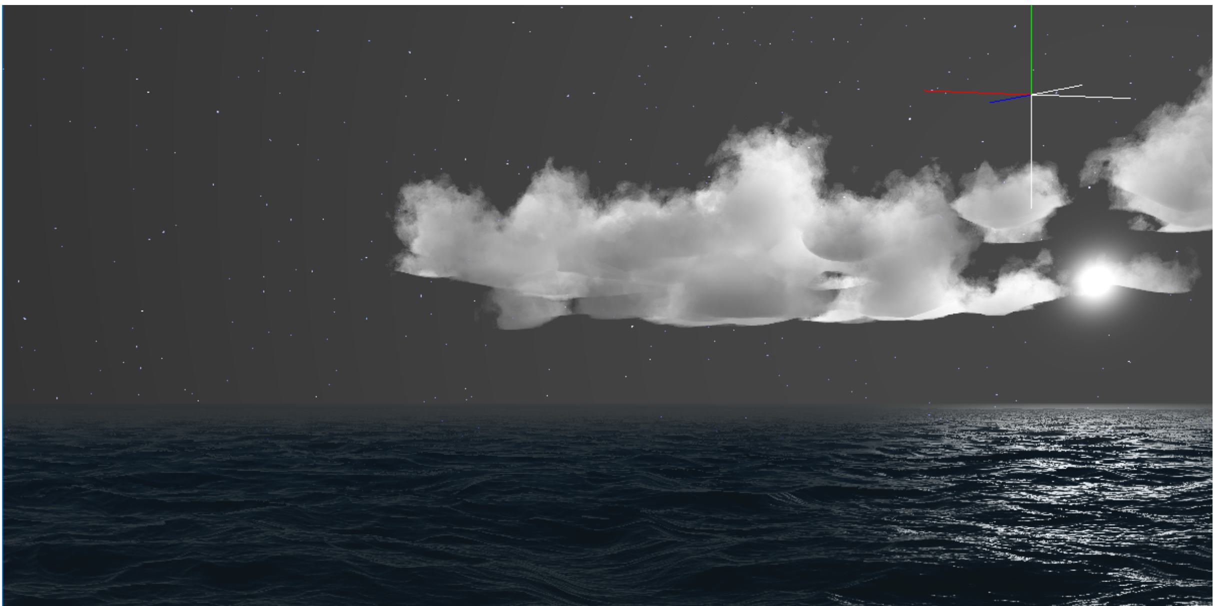


Figure 5.8. Full moon over the ocean.

Figure 5.8 is a sample of a night scene with the moon lighting the clouds with specific parameters of the phase and light colour.

5.6 Summary

This chapter is intended to introduce the lighting system used in the present thesis by referencing the main authors on whose work the research is based and the author contributions to the state of the art in the volumetric cloud rendering field. Therefore, this discussion illustrates the base algorithm for lighting and shading and the mathematical theory development about the implemented model. The chapter also explains the NDT algorithm used to simplify the raytracing in addition to its application in the precomputed light parallelization. Finally, several image samples of enlighted clouds in different day stages are presented.

CHAPTER 6

Cloud shape improvement

THis chapter presents four different methods to define the geometry of the cloud to fulfil different requirements from an artistic point of view. These methods range from purely mathematical random generation to user-defined 3D meshes modelled in an editor. Thus, in Section 6.1 is explained the basic formulas to generate a Gaussian cumulus cloud along with a complete case study of a cumulus cloud from a statistical point of view. Later, in Section 6.2 is introduced the approach of this thesis to emulate the level of condensation in cumulus clouds. A new strategy to generate fractal clouds by using the spheroid distribution is described in Section 6.3. The optimization of the system performance from the application of metaballs with a simple modification of Algorithm 4.1 is detailed in Section 6.4. Finally, a novel technique to generate a cloud resembling known forms based on smooth ellipsoid adjustment and the collision detection formula is introduced in Section 6.5.

6.1 Gaussian cumulus cloud generation

6.1.1 3D Gaussian cloud generation

The proposed algorithm accomplishes cumulus generation by using a three-dimensional Gaussian distribution of basic pseudospheroid primitives already seen in Section 4.3. The thesis implementation uses a Gaussian distribution with clamping to generate the position of the pseudospheroids as it is done for the particles in the method by Huang et al. [Hua+08]. Unlike their work, the number of primitives to produce a realistic cloud is much smaller than in a particle system. Typically the model uses 35 pseudospheroids while the particle system requires thousands of them.

Let P be the position of the pseudospheroid, C the center of the cloud, and $\mathscr{N}$ a random variable with normal distribution, the base equation is:

$$\begin{cases} P_x = C_x + \sim \mathcal{N}(\mu_x, \sigma_x) \\ P_y = C_y + \sim \mathcal{N}(\mu_y, \sigma_y) \\ P_z = C_z + \sim \mathcal{N}(\mu_z, \sigma_z) \end{cases} \tag{6.1}$$

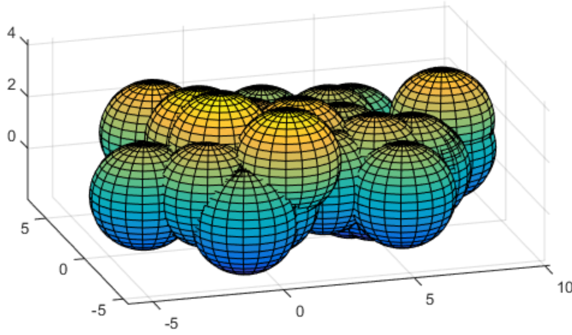


Figure 6.1. A cumulus plot using Equation 6.3 with $N_x(0{,}3)$, $N_y(0{,}0.9)$, $N_z(0{,}3)$ and $\varepsilon_2 = 2.0$.

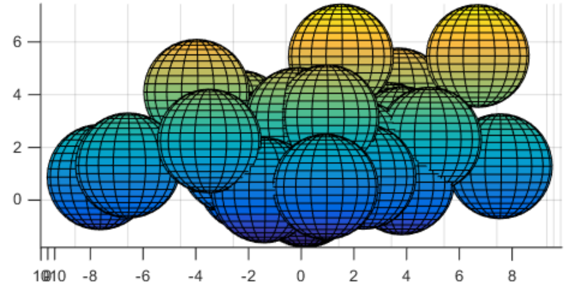


Figure 6.2. Another cumulus plot using Equation 6.3 with $N_x(0{,}4.5)$, $N_y(0{,}2.9)$, $N_z(0{,}4)$ and $\varepsilon_2 = 2.0$.

Table 6.1 shows the typical μ and σ of the normal distribution used in each of the axes. The previous values must be added to the center of the cumulus to place the cloud into the scene.

Axis	μ	σ	Clamped
X	μ_x	σ_x	$[-k_1\sigma_x, k_2\sigma_x]$
Y	μ_y	σ_y	$[\mu_y, t\sigma_y]$
Z	μ_z	σ_z	$[\text{-}m_1\sigma_z, m_2\sigma_z]$

Table 6.1. Typical parameters for the proposed Gaussian equations.

The radii of the pseudospheroids depend on the distance from their center (P) to the center of the cloud (C) according to a Gaussian function, as shown in Figures 6.1 and 6.2. Two variants of the proposed distribution can be used to compute the magnitude of the radius as described in Equations 6.2 and 6.3:

$$radius_i = \frac{\varepsilon_1}{|P_x - C_x| + |P_z - C_z| + 1.0} \tag{6.2}$$

$$radius_i = \varepsilon_2 \cdot \left(1.0 - 0.1\sqrt{((P_x - C_x)/2\sigma_x)^2 \cdot ((P_y - C_y)/2\sigma_y)^2 \cdot ((P_z - C_z)/2\sigma_z)^2}\right) \tag{6.3}$$

where ε_i represents the maximum possible radius of each sphere in the corresponding variant. The experiment used the constants $\varepsilon_1 = 15.0$ and $\varepsilon_2 = 2.5$ for real cumulus generation.

This thesis introduces another improvement to hit the performance target with a pre-processing algorithm that reduces the list of spheroids by removing those that are within the range $[3\sigma_x/4, \sigma_y/3, 3\sigma_z/4]$ which creates a sort of hollow cloud. This filter causes the removal of $\sim 20\%$ of spheroids in the nucleus of the cloud, which yields a $\sim 30\%$ average improvement in frame rate without affecting the look of the cloud. In addition to the prior filtering, the pre-processing algorithm removes spheres contained inside other spheres by comparing the differences in radii (r_i, r_j) with the distance between their centers (x_i, y_i, z_i) and (x_j, y_j, z_j) using Equation 6.4.

$$\forall i \forall j |r_i - r_j| \geq \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \tag{6.4}$$

The $O(n^2)$ complexity of the previous computation is not a big burden due to the reduced number of spheres required in the present Gaussian model that do not require, in this case, real-time processing.

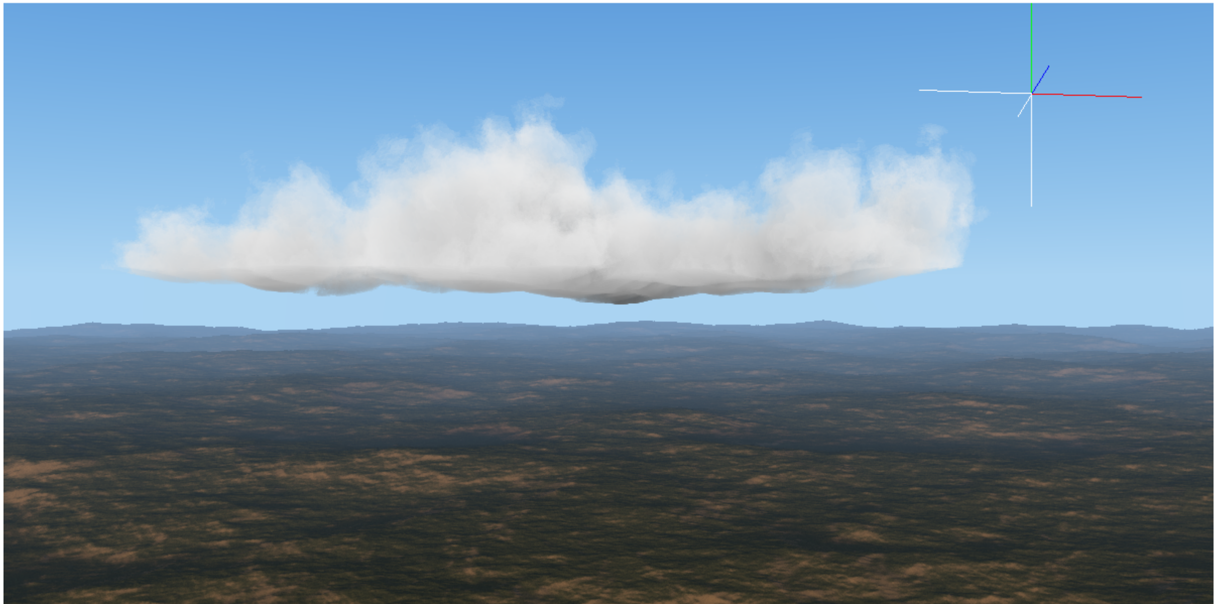


Figure 6.3. A rendered Gaussian cumulus cloud with $N_x(0,3.5), N_y(0,0.9), N_z(0.3.0)$ and $\varepsilon_2 = 1.5$.

6.1.2 Case statistical analysis

Let us consider a generated cloud sample with $N(0,2)$ distribution $\forall_x,\forall_y,\forall_z$ with the data shown in Table 6.2:

P_x	P_y	P_z	R
4.52892	0	-6	2.18678
0.840693	2.76951	0.360271	2.32276
-0.344144	0	-3.07531	2.37106
2.13971	0	0.686463	2.40637
3.81488	0	-0.405287	2.34015
-0.416573	0	0.21741	2.48042
-2.87359	1.57103	0.923407	2.34045
...	...	...	...

Table 6.2. The generated data for the mentioned cloud.

After processing the data with the R statistical application [1], the histogram of Figure 6.4 for the P_x coordinate is obtained. A pseudo-Gaussian distribution can be observed in the blue plot centred in the range [-1,0].

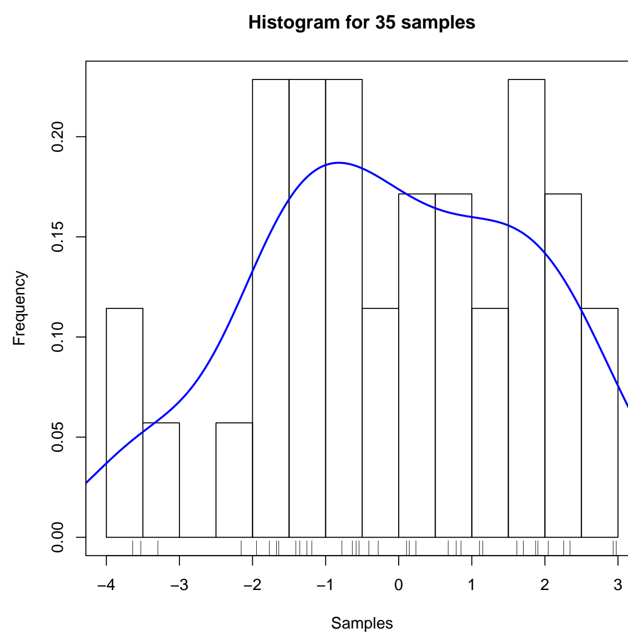


Figure 6.4. The generated histogram for the P_x coordinate of Table 6.2.

However, for the highest precision, we calculate the maximum likelihood estimators [Ald97]:

[1] https://www.r-project.org/

mean	sd
-0.09645557	1.78332609
(0.30143713)	(0.21314824)

Table 6.3. Estimator result using *fitdistr* in R with its corresponding approximation errors.

As demonstrated in Table 6.3, the estimator calculus reveals that the distribution satisfies $N(0,2)$.

Finally, to compare the distribution of the current generated cloud with other distributions, we make use of Q-Q diagrams. For this purpose, an envelope restriction of 95% has been applied in all graphs, and the results are shown in Figure 6.5. The normal plot for the P_x coordinate is the most accurate approximation for the analysed data, producing the least number of samples (black circles) outside the envelope (dotted blue lines).

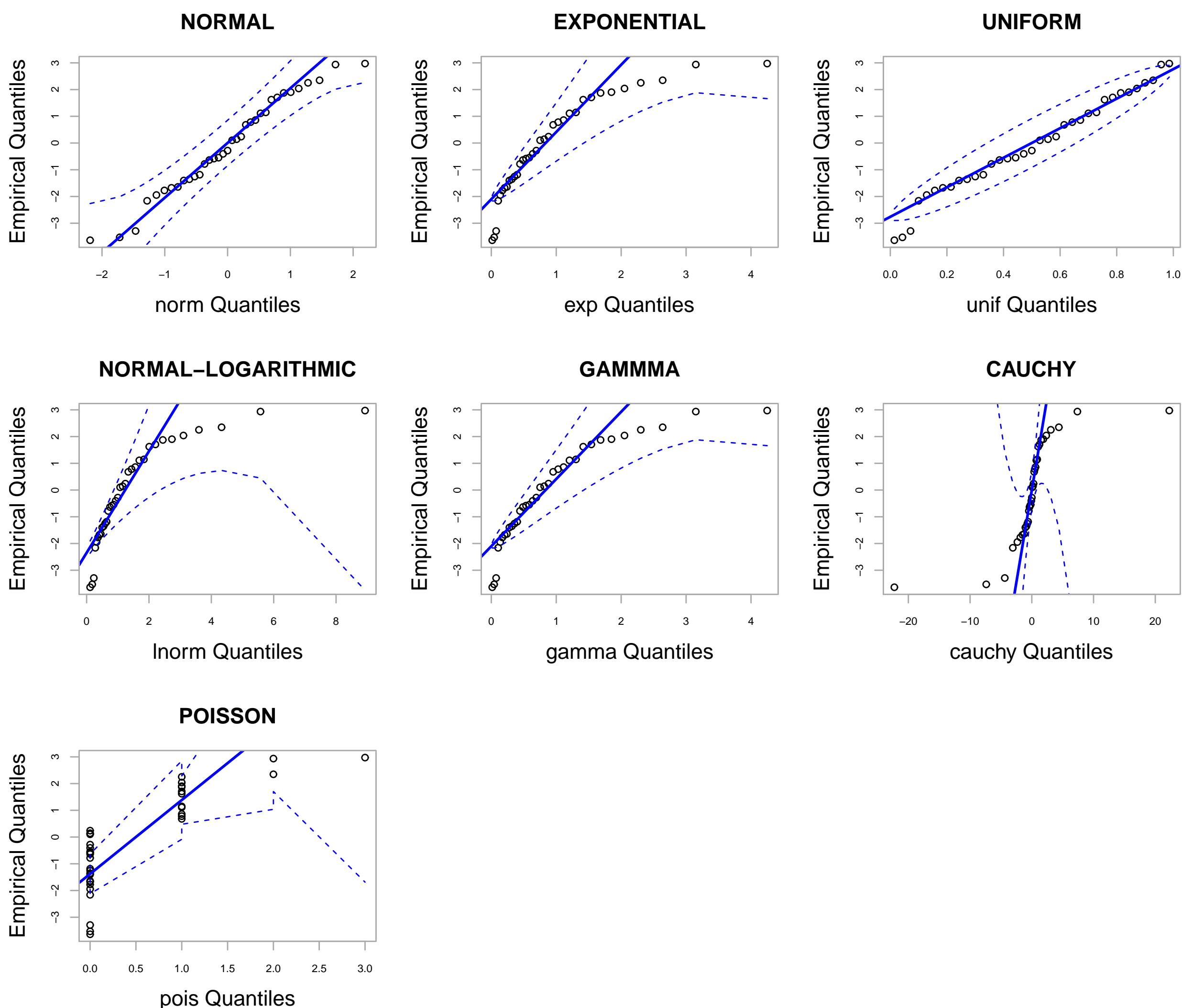


Figure 6.5. Q-Q diagram for the P_x coordinate of Table 6.2 plotted in R.

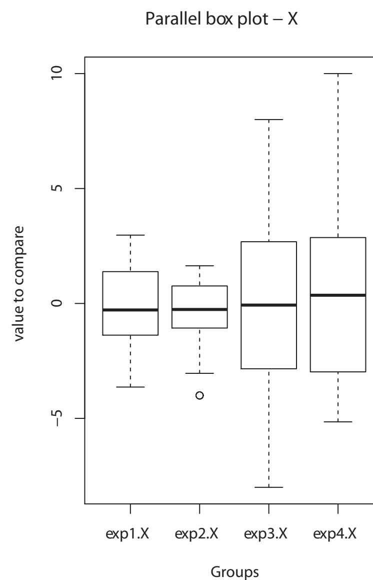


Figure 6.6. Box-and-whisker plot of the P_x coordinate with the other three samples.

As seen in Figure 6.6 a strong overlap between the boxes occurs for samples 1,2,3 and 4, which can indicate that they follow the same distribution. Finally, we confirm the model accuracy by applying the nonparametric *Kruskal-Wallis* test with two hypotheses:

- H_0: All the distributions are identical.
- H_1: The distribution of at least one of the sample sets tends to produce values greater than the minimum of the distribution of another set of samples.

After applying the test, we obtain that:

Kruskal-Wallis chi-squared = 0.69665, df = 3, **p-value = 0.874**.

Thus, we claim that when the p-value is greater than 0.001, we accept the null hypothesis, and therefore the stochastic model in C++ generates correct clouds.

6.2 Level of condensation

The level of the condensation effect of a typical cumulus cloud as seen in Section 2.7 can be achieved using the contents of the precomputed light voxel values. Thus, the researched algorithm in this thesis uses the previous values to adjust the bottom side of the cloud from a specific height that has been previously reduced in the host application. Figure 6.7 illustrates the cited case. Assuming that Y is the ordinate axis, the raymarcher passes only the rays satisfying Equation 6.5.

$$(!bFlat \vee (rayPos.y > gridMin.y + voxel[rayPos] \times \delta)) \implies render \quad (6.5)$$

where δ is the adjustment factor.

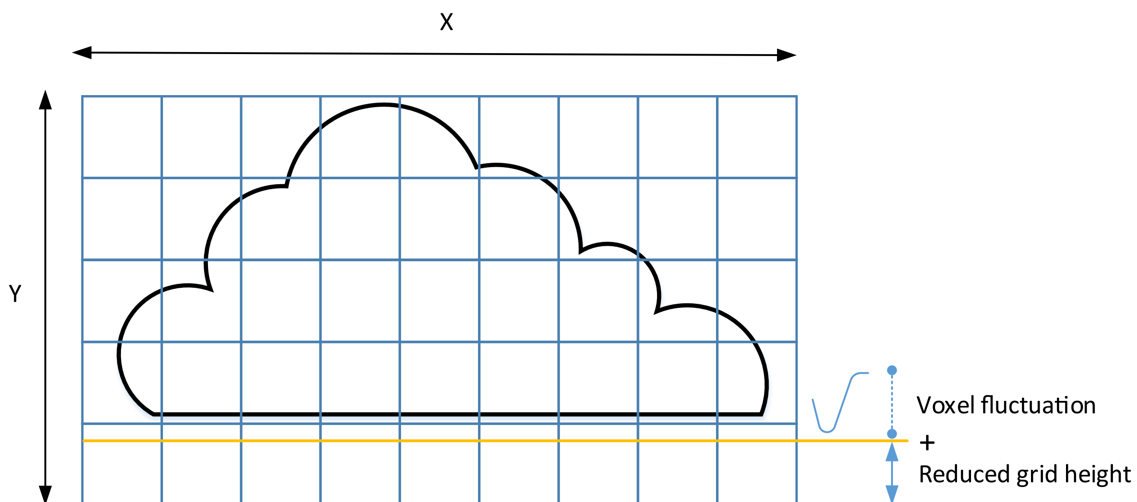


Figure 6.7. Flat cumulus by varying the voxel precomputed light value relative to the reduced height of the grid.

6.3 Fractal cumulus generation with L-systems

L-systems, also called Lindenmayer systems, were originally developed in 1960 by the biologist *Aristid Lindenmayer* to describe the fractal evolution of plants and other natural phenomena. In collaboration with *Przemyslay Prusinkiewicz*, a large collection of impressive images of nature was collected [PL96]. These systems consist of a set of grammar rules with a defined alphabet which are recursively called, starting from an axiom production. Subsequent calls generate a string of symbols which denote a specific meaning. The proposed research uses L-system grammars to generate a wide variety of cloud shapes. A deep study on the L-system is also explained in [Jon01] with exhaustive algorithms for clouds and plants, respectively. The approach of this thesis differs from that of Kang et al. [KPK15] in the addition of a proportional random factor in the generation of the sphere radius and the distance between primitives in each recursive call, as explained in Figure 6.8. Another difference from the cited work is

the use of volumetric rendering and the proposed density function (Equation 4.9). The 3D algorithm for L-system generation of spheres is referred to in the website of the *Department of Computing for Neurobiology at Cornell University*, which contains a complete MATLAB explanation and sources.

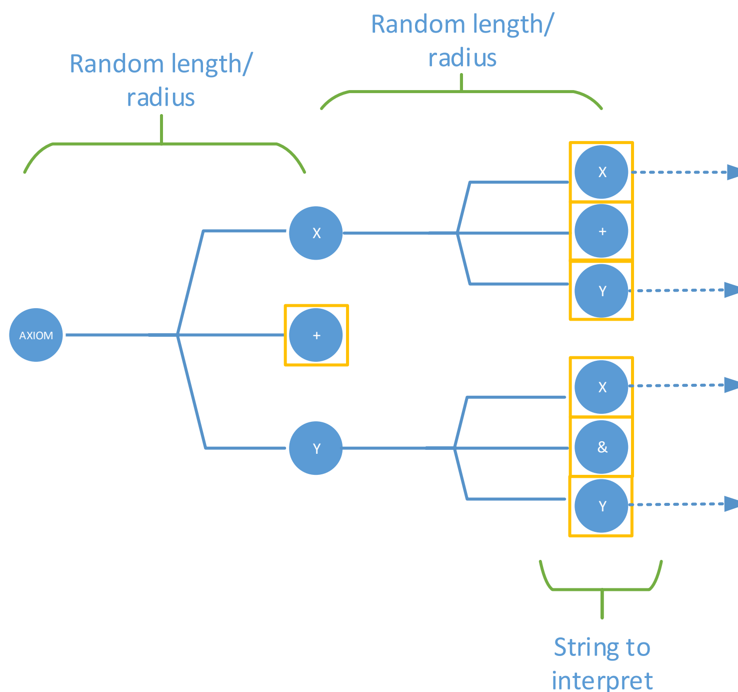


Figure 6.8. After each recursive call, the interpreter generates a new proportional random radius and length for the primitives. This produces more natural and impressive cumuliforms.

Before calling the GLSL image render, the proposed solution uses an algorithm coded in C++ running on the host as a tiny interpreter for the recursive generation of spheres according to axiom and rules. For example, in the call to the following Backus-Naur form (BNF) grammar, the implemented C++ algorithm will generate a string of "X","Y","+" and "&" symbols:

$$\begin{aligned} Axiom &\rightarrow X \\ X &\rightarrow X + Y \\ Y &\rightarrow X \,\&\, Y \end{aligned} \tag{6.6}$$

In the Equation 6.6 " + " means turn left with a δ angle using a rotation matrix, and " & " means pitch down with a δ angle by using a rotation matrix. As a result, Figures 6.9, 6.10 were generated using different angles and iterations.

The corresponding lexical analyser must insert a sphere of a proportional uniform random radius and length in the current branch step for each of the X and Y items found, and then apply the rotation angle given by the " + " and " & " operators. Hence, the string produced by the equation 6.6 will generate 128 spheres according to the L-system grammar shape and will be ready to transfer to the OpenGL shader for real-time animation.

The thesis L-system grammar derivative is suitable for generating jet streams and a sort of cloud called *Castellanus*, as cited by Häckel [Häc06] due to its elongated and crenelated shape. The advantage of this interpreter is that it allows a formal definition of clouds for designers to generate a variety of original cloud shapes for computer games and digital art.

Figure 6.9. Iterations = 7, $\delta = 10^{\circ}$. The lower the δ angle, the thinner and more strange the cloud shape becomes, so the cloud looks like a 3D spiral. This is caused by the recursive turning operators in the grammar productions that respond to the director rotation angle and the uniform random sphere radius.

Figure 6.10. Again, using exactly the same grammar and iterations with a yet larger angle, for instance $[50^{\circ},100^{\circ}]$, the resulting grammar derivation generates dense cumulus formations.

6.4 Cumuliforms with optimized metaballs

The metaballs technique proposed by Blinn [Bli82] for the creation of organic-looking n-dimensional objects could be also applied to generate cumulus formations with very high performance and render quality. The proposed model differs from the Dobashi approach [Dob+99] in that it uses optimized equations and a mean calculated in the CPU:

$$f(x,y,z) = \gamma = \sum_{i=1}^{spheres} \frac{{r_i}^2}{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2} \geq 1 \tag{6.7}$$

where x,y,z is the raymarching position, r_i is the current sphere radius and x_i,y_i,z_i is the sphere center.

$$\bar{r}^2 = \sum_{i=1}^{N=spheres} \frac{{r_i}^2}{N} \tag{6.8}$$

and where

$$\rho = fbm(rayPos) \cdot \bar{r}^2 \cdot \phi \tag{6.9}$$

Equation 6.8 is pre-calculated in the CPU only once, and Equations (6.7) and (6.9) are used in the GPU shader algorithm. ϕ is a constant to adjust the fading effect in the edges of the cloud and $rayPos$ is the Euclidean straight line point of the ray in R^3 used to evaluate the fBm noise explained in Section 4.2. It usually has a value of 0.6 in the tests of the present thesis.

This method has very high performance and achieves a remarkable number of frames-per-second. Besides this, render quality is sufficient, as shown in Figure 6.11, but lower than that generated by Algorithm 4.1.

6.5 Cloud generation from 3D meshes

6.5.1 Basic operations

This section explains the last geometry innovation for real-time cloud rendering based on using three-dimensional mesh editors like *Blender*, *3D Studio*, or *Maya*. Making smoke or clouds with known shapes is a complex task that requires algorithms and mathematical optimization in addition to a suitable 3D model before rendering. The user may use the GPL *Blender* application as a geometry editor for several models used in this research. Effective real-time rendering needs a previous decimation of big meshes up to a level of 1000 triangles or less. Bigger quantities require more powerful GPUs and do not pay off since the spatial definition of smoke does not allow high detail. The key idea of this algorithm is replacing triangles with ellipsoids to achieve more real and suggestive cumuliforms. To improve performance, the ellipsoids are pre-calculated in the host and then rendered on the GPU, applying the same

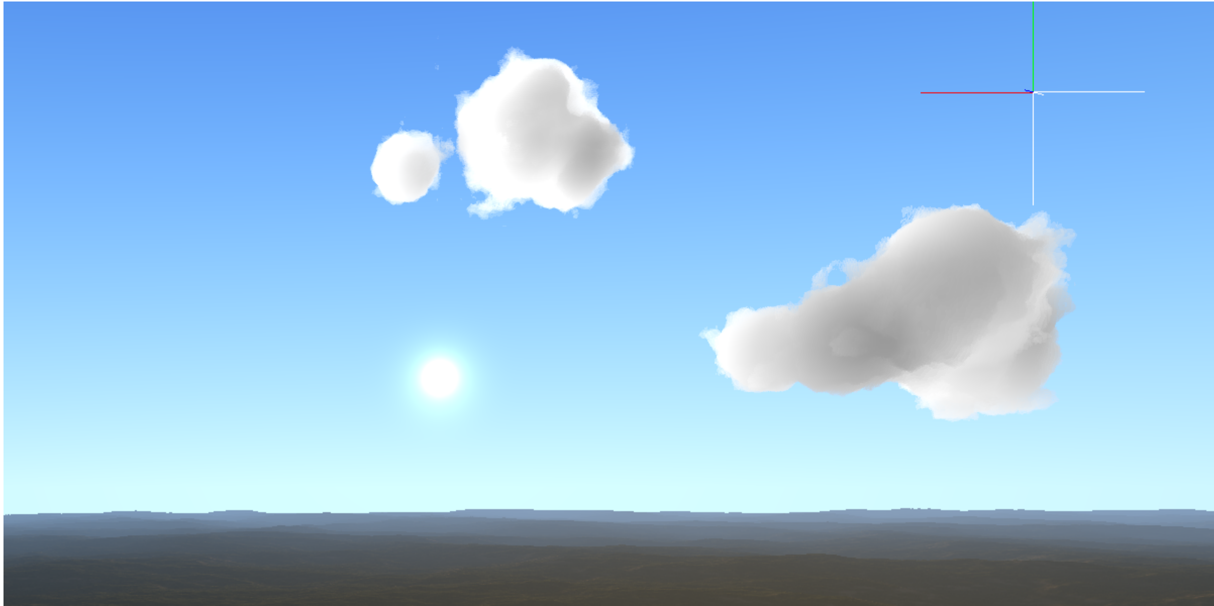


Figure 6.11. Two cumulus formations by using Gaussian distribution and optimized metaballs. Just six spheres were used to render the samples by randomizing the sphere radius.

techniques explained in Chapters 4 and 5. Initially an ellipsoid is centered at the barycenter of each triangle. This is achieved by scaling the triangle by a specified amount from the barycenter without translating it. The maximum distance between each scaled vertex and the barycenter is used to calculate a circumscribing ellipsoid with radius (a, b, c). This approach is effective and accurate, as explained in Figures 6.12 and 6.13.

Let P_1, P_2 and P_3 be the vertices of a triangle. The proposed model defines Barycenter (B) as:

$$B = \left(\sum_{i=1}^{3} \frac{x_i}{3}, \sum_{i=1}^{3} \frac{y_i}{3}, \sum_{i=1}^{3} \frac{z_i}{3} \right) \tag{6.10}$$

and scale as:

$$P_i' = (P_i - B) \cdot scale + B \tag{6.11}$$

$$\begin{aligned} radius_a &= \|(B - P_1')\| \\ radius_b &= \|(B - P_2')\| \\ radius_c &= \|(B - P_3')\| \end{aligned} \tag{6.12}$$

where P_i' is the scaled triangle vertices (x,y,z) from 1 to 3.

An optimization of the previous algorithm could be made by performing a R^3 rotation of the direction vector, with the maximum radius of the ellipsoid over the direction vector of the maximum distance from the triangle vertices to the barycenter. The solution requires the application of the principles of *Rodrigues' Rotation Formula* for this issue. This solution provides more accurate geometry with the cloud description, and a softer shape for the model. This approach may be considered as a real-time filter for mesh geometry, producing rough

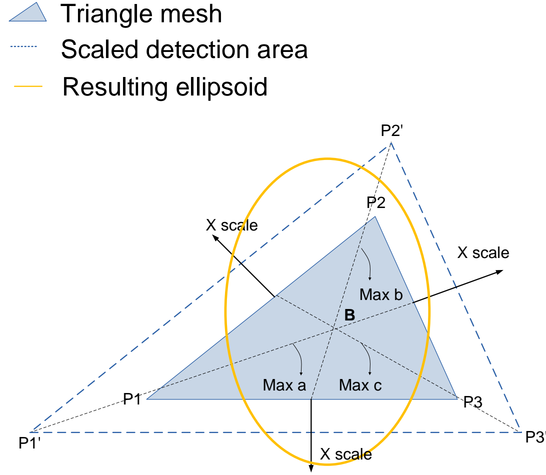


Figure 6.12. Ellipsoid scaling process. The proposed algorithm multiplies each triangle vertex (P_1,P_2,P_3) by a factor that typically falls in the range (0.1,2], once the barycenter (B) is calculated. Afterwards, the shader algorithm uses the maximum distance from the barycenter to the scaled triangle vertex as relation for density estimation through the ellipsoid/ray collision equations.

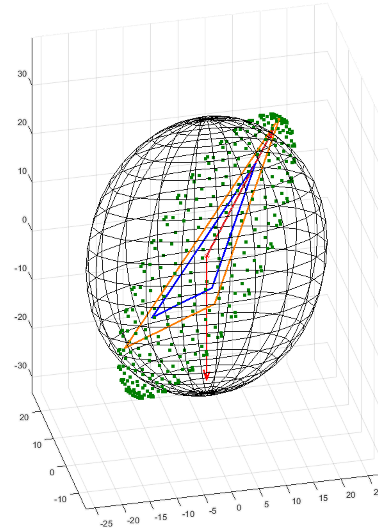


Figure 6.13. After rotation. As seen in the image above, showing the original ellipsoid in black and the resulting one in green, the previous equations allow the overlapping of the R^3 direction vectors. Hence and according to the direction of the larger triangle vertex, the algorithms produce the resulting rotation.

results. Therefore, a continuation of the previous formulas is as shown:

$$if \begin{cases} max(radius_a) & dir\vec{T}ria = P_1' - B \\ max(radius_b) & dir\vec{T}ria = P_2' - B \\ max(radius_c) & dir\vec{T}ria = P_3' - B \end{cases} \tag{6.13}$$

$$if \begin{cases} max(radius_a) & dir\vec{E}llip = (radius_a + B_x, B_y, B_z) - B \\ max(radius_b) & dir\vec{E}llip = (B_x, radius_b + B_y, B_z) - B \\ max(radius_c) & dir\vec{E}llip = (B_x, B_y, radius_c + B_z) - B \end{cases} \tag{6.14}$$

(1) Axis and angle using cross product and dot product:

$$x = \frac{dir\vec{E}llip \times dir\vec{T}ria}{\|dir\vec{E}llip \times dir\vec{T}ria\|} \tag{6.15}$$

$$\theta = cos^{-1}\left(\frac{dir\vec{E}llip \cdot dir\vec{T}ria}{\|dir\vec{E}llip\| \cdot \|dir\vec{T}ria\|}\right) \tag{6.16}$$

(2) Rotation matrix using exponential map:

$$R = e^{A\theta} = I + sin(\theta) \cdot A + (1 - cos(\theta)) \cdot A^2 \quad (6.17)$$

(3) A is a skew-symmetric matrix corresponding to x:

$$A = [x]_x = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \quad (6.18)$$

Equations 6.15, 6.16, 6.17 and 6.18 are pre-calculated in the CPU side only once, and solely the R rotation matrix is passed to the GPU for real-time rendering.

Thanks to this algorithm, the overall performance is not reduced even in lower speed GPUs as it is demonstrated in the benchmarks sections 8.2.1 and 8.2.2.

Decimation of the mesh to 300–700 triangle faces is required to increase performance and provide better conformance to the cloud shape. For instance, the hand mesh wireframe is decimated to 354 faces before rendering as seen in Figure 6.14. A 95% decimation still produces good visualization and performance (Figures 6.15 and 6.16). Lower decimations may be considered depending on the hardware used.

Figure 6.14. A hand mesh transformed into a soft 3D cloud. The final result is successfully optimized for the real-time GPU algorithm animation and rendering.

Figure 6.15. A rabbit mesh with 370 triangles. 80% decimation has been performed on this mesh, reducing the number of triangles from 1850 to 370 to achieve a suitable real-time performance.

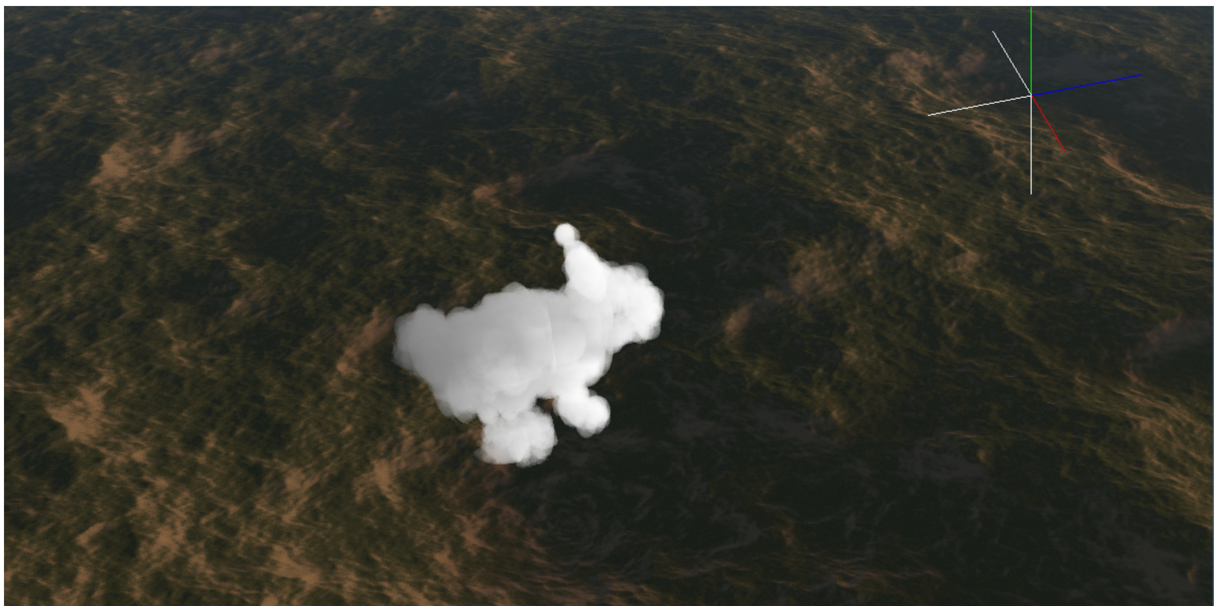


Figure 6.16. The rabbit mesh blended with the ground of the landscape. The rotational filter allows to recognize the mesh correctly.

6.5.2 Ray-ellipsoid intersection approach

This section is intended to analyse the mathematical artefacts used to detect the intersection between the ray departing from the frame buffer with the rotated pseudoellipsoid. As a consequence, it is important to recall the Euclidean parametric straight line equation and the ellipsoid implicit equation, corresponding to Equations 6.19 and 6.20, respectively.

$$R \equiv \begin{cases} x = x_0 + \lambda \vec{v}_x \\ y = y_0 + \lambda \vec{v}_y \\ z = z_0 + \lambda \vec{v}_z \end{cases} \tag{6.19}$$

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{6.20}$$

Since the straight line follows Rodrigues' rotation formula, we need to specify the following transformation:

$$M = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} x - c_x \\ y - c_y \\ z - c_z \end{bmatrix} \tag{6.21}$$

where c_x, c_y and c_z in Equation 6.21 are the barycenter coordinates.

Therefore, if we substitute M into Equation 6.20 and solve λ, we obtain the discriminant:

$$\frac{M(1)^2}{a^2} + \frac{M(2)^2}{b^2} + \frac{M(3)^2}{c^2} = 1 \tag{6.22}$$

$$\Delta = \frac{\left(\frac{(2\,k11\,vx + 2\,k12\,vy + 2\,k13\,vz)\,(k11\,(cx - x0) + k12\,(cy - y0) + k13\,(cz - z0))}{a^2} + \frac{(2\,k21\,vx + 2\,k22\,vy + 2\,k23\,vz)\,(k21\,(cx - x0) + k22\,(cy - y0) + k23\,(cz - z0))}{b^2} + \frac{(2\,k31\,vx + 2\,k32\,vy + 2\,k33\,vz)\,(k31\,(cx - x0) + k32\,(cy - y0) + k33\,(cz - z0))}{c^2} \right)^2}{4} - \left(\frac{(k11\,vx + k12\,vy + k13\,vz)^2}{a^2} + \frac{(k21\,vx + k22\,vy + k23\,vz)^2}{b^2} + \frac{(k31\,vx + k32\,vy + k33\,vz)^2}{c^2} \right) \left(\frac{(k11\,(cx - x0) + k12\,(cy - y0) + k13\,(cz - z0))^2}{a^2} + \frac{(k21\,(cx - x0) + k22\,(cy - y0) + k23\,(cz - z0))^2}{b^2} + \frac{(k31\,(cx - x0) + k32\,(cy - y0) + k33\,(cz - z0))^2}{c^2} - 1 \right) \tag{6.23}$$

Finally, we use Δ to detect ray-ellipsoid intersection and obtain λ_{in} and λ_{out} consecutively.

6.6 Summary

This chapter is an overview of the accomplished work around the cloud shape improvement as a new approach for generating efficient cloud rendering. Its contents address new formulas to recreate Gaussian cumulus clouds with a complete statistical study of a generated cloud. In addition to the Gaussian cumulus clouds, the chapter describes the approach to emulate the level of condensation by flattening the bottom side of the cumulus cloud using the GLSL shader resources. The chapter also reveals the researched method for fractals and the metaball optimization technique as a new perspective for cloud shape improvement. The present chapter finalizes with a wide explanation of the novel technique to generate pareidolic [2] effects from decimated 3D meshes using the *Rodrigues' rotational formula* to smooth the cloud form and the mathematical development to detect ray-ellipsoid collision in this new model.

[2] A situation in which someone sees a pattern or image of something that does not exist, for example a face in a cloud (Cambridge Dictionary).

CHAPTER 7

Cloud dynamics simulation

THE main purpose of this chapter is recreate the main techniques and the researched models to simulate cloud advection and convection movement in the atmosphere. Therefore, in Section 7.1 is reviewed the theoretical background about fluid mechanic physics in computer simulations and the exposition of CUDA parallelization principles in the fluid class implementation. The simplified novel technique of cloud animation and deformation using the parameters provided by the fluid engine class is graphically explained in Section 7.2. Section 7.3 illustrates the proposed model for the metamorphosis between two mesh clouds with a previous introduction to the referenced bibliography about this matter.

7.1 Fluid simulation background

7.1.1 Introduction

We have already introduced the Navier-Stokes equations in Section 2.11. Now, we address the stable fluid method, which is used to solve these equations in a computer graphics simulation. As described in [Ama09], the stable fluid method requires an implicit linear system solver such as, for example, the Jacobi, Gauss-Seidel or conjugate gradient methods.

As explained in Section 2.11, the motion of a viscous fluid can be demonstrated with the Navier-Stokes equations, which are a set of partial differential equations that state a relation between the pressure, velocity and forces during a specific period of time. Therefore, the fluid properties are described in relation to the density and velocity. As seen before, the Navier-Stokes equations are based on Newton's Second Law of Motion:

$$F = m \cdot a \tag{7.1}$$

where m is the mass, a is the particle acceleration, and F is the resulting force.

There are other alternatives to describe fluid motion, such as the Euler equations used in [Har03]; however, the Navier-Stokes equations are the most advanced ones, as seen in the following.

The Navier-Stokes equations used in the present thesis state that:

$$\nabla v = 0 \tag{7.2}$$

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u + v\nabla^2 u + F \tag{7.3}$$

$$\frac{\partial p}{\partial t} = -(u \cdot \nabla)\rho + k\nabla^2 \rho + S \tag{7.4}$$

where u is the velocity field, v is the viscosity of the fluid and F is the external force applied to the velocity field. ρ is the field density, k describes the density diffusion rate, S is the external force applied to the density field, and $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$ represents the gradient.

Equation 7.2 ensures mass conservation and initializes the velocity field to zero. Equations 7.3 and 7.4 describe the evolution of the velocity and the density, respectively.

This thesis uses the [Sta03] approach to simulate cloud motion; in addition, the density equation 7.4 is not required in its approach due to the use of the novel method of cloud guide points as explained in Section 7.2.

The basic algorithm 7.1 based on the work of [Sta03] has been modified in the following manner as cited in [JG19]:

```
1 while simulating do
2
3     Retrieve UVW force equations
4     Add force
5     Diffuse
6     Project
7     Advect
8     Apply to guide points
9 end
```

Algorithm 7.1: Execution flow of the thesis fluid simulator.

- **Add force:** The process consists in adding a 3D force F (Equation 7.3) to the velocity field in each grid cell. For each grid cell, the new velocity is:

$$u = u_0 + \Delta t \times F. \tag{7.5}$$

where u is the velocity, u_0 it the previous value of u, Δt is the time increment and F is

the applied force (wind).

- **Advection:** The advection function describe the transport of clouds, which is the resulting velocity of the fluid when moving. For a better understanding of advection, let us consider that each grid cell defines a fluid particle as illustrated in Figure 7.1:

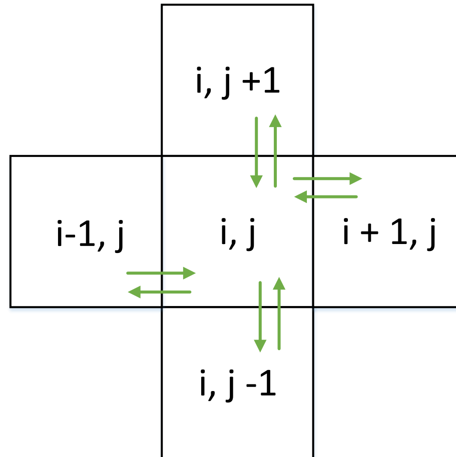


Figure 7.1. The velocity vector components are stored in a 3D grid of cells.

The advection can be calculated either with an explicit method such as the Euler method, a midpoint method such as Runge-Kutta or an implicit method such as [Sta03].

If an explicit method is used, Equation 7.6 results:

$$r(t+\delta t) = r(t) + u(t)\delta t \tag{7.6}$$

where $r(t)$ is the particle position and δt is the elapsed time while moving along the velocity field u. The problem is that when $u(t)\delta t$ is greater than the size of the grid cell, the simulation fails. Nevertheless, [Sta03] overcomes this problem with an implicit solution approach as explained in Equation 7.7:

$$q(x,t+\delta t) = q(x-u(x,t)\delta t, t) \tag{7.7}$$

where q is a quantity carried by the fluid, e.g., velocity, temperature, or density.

«With stable fluid methods, the trajectory of the particle from each grid cell is traced back in time, to its former position. This approach is also referred as semi-Langrangian advection». [Ama09].

- **Diffusion**

The diffusion is represented by the $\nu\nabla^2 u$ term in Equation 7.3, where the viscosity is the fluid internal resistance to flow. The resistance is caused by the diffusion of the momentum, such as the dissipation of velocity.

As with advection, the diffusion can be approximated either with explicit or implicit methods.

The explicit approach of diffusion is described in Equation 7.8:

$$u(x,t+\delta t) = u(x,t) + v\delta t \nabla^2 u(x,t) \quad (7.8)$$

where ∇^2 is the discrete form of the Laplacian operator, δt is the time increment and v is the viscosity. This formulation also fails during the simulation.

The implicit version obeys the explicit Equation 7.9:

$$(I - v\delta t \nabla^2)u(x,t+\delta t) = u(x,t) \quad (7.9)$$

where I is the identity matrix. This equation, called the Poisson equation, remains stable for arbitrary time increments and viscosities and can be solved by iterative solvers such as Jacobi relaxation, Gauss-Seidel relaxation and conjugate gradient. This thesis opts for the first technique because the ontogenetic implementation does not require a lot of precision, so an efficient and reliable simulation is enough for the standard industry purposes.

7.1.2 Fluid class parallelization

We consider the fluid simulator as a 3D grid with cell interactions with their neighbours as seen in Figure 7.1. The CUDA implementation is similar to that explained in Section 5.4 and cited in [JG19] with the distribution of the threads (T_x, T_y, T_z) according to the *maxThreadsPerBlock* device capability, which is $T_x \times T_y \times T_z = 1024$, so the distribution is $T_x = 16, T_y = 8, T_z = 8$. The dimensions of the block are:

$$Block_{x,y,z} = \left\lceil \frac{Dim_{x,y,z} + 2}{T_{x,y,z}} \right\rceil \quad (7.10)$$

where $Dim_{x,y,z}$ are the dimensions of the voxel grid.

As the approach of this thesis is based on [Sta03], the cloud movements are simulated inside a relocatable grid tunnel of $M \times N \times O$ corresponding to x,y,z dimensions where the wind force vectors are applied and calculated.

According to the modified Algorithm 7.1 a CUDA parallel class has been developed by replacing Stam's serial triple for a loop of 8-10 lines as:

```
1 void FluidCPU::advect(int b, float* d, float* d0, float* u, float* v, float* w
      , float dt)
2 {
3   int i, j, k, i0, j0, k0, i1, j1, k1;
4   float x, y, z, s0, t0, s1, t1, u1, u0, dtx, dty, dtz;
5
6   dtx = dty = dtz = dt * MAX(MAX(dimM, dimN), MAX(dimN, dimO));
7
8   for (i = 1; i <= dimM; i++) {
9     for (j = 1; j <= dimN; j++) {
10      for (k = 1; k <= dimO; k++) {
11        x = i - dtx * u[IX(i, j, k)]; y = j - dty * v[IX(i, j, k)]; z = k -
              dtz * w[IX(i, j, k)];
12        if (x < 0.5f) x = 0.5f; if (x > dimM + 0.5f) x = dimM + 0.5f; i0 = (
              int)x; i1 = i0 + 1;
13        if (y < 0.5f) y = 0.5f; if (y > dimN + 0.5f) y = dimN + 0.5f; j0 = (
              int)y; j1 = j0 + 1;
14        if (z < 0.5f) z = 0.5f; if (z > dimO + 0.5f) z = dimO + 0.5f; k0 = (
              int)z; k1 = k0 + 1;
15
16        s1 = x - i0; s0 = 1 - s1; t1 = y - j0; t0 = 1 - t1; u1 = z - k0; u0 =
              1 - u1;
17        d[IX(i, j, k)] = s0 * (t0*u0*d0[IX(i0, j0, k0)] + t1 * u0*d0[IX(i0, j1
              , k0)] + t0 * u1*d0[IX(i0, j0, k1)] + t1 * u1*d0[IX(i0, j1, k1)]) +
18          s1 * (t0*u0*d0[IX(i1, j0, k0)] + t1 * u0*d0[IX(i1, j1, k0)] + t0 *
                u1*d0[IX(i1, j0, k1)] + t1 * u1*d0[IX(i1, j1, k1)]);
19      }
20    }
21  }
22 }
```

Listing 7.1. Original serial code version for advection.

with the following code calling the CUDA parallel kernel in blocks and threads.

```
1 void FluidCUDA::advect(float d[][N + 2][O + 2], float d0[][N + 2][O + 2],
      float u[][N + 2][O + 2], float v[][N + 2][O + 2], float w[][N + 2][O + 2],
      float dt)
2 {
3   dim3 block(numBlocksX, numBlocksY, numBlocksZ);
4   dim3 thread(THREADS_X, THREADS_Y, THREADS_Z);
5
6   float dtx, dty, dtz;
7
8   dtx = dty = dtz = dt * MAX(MAX(M, N), MAX(N, O));
9
10  kernelAdvect <<< block, thread >>> (d, d0, u, v, w, dt, dtx, dty, dtz);
11 }
```

Listing 7.2. Kernel call.

```
1 __global__ void kernelAdvect(float d[][N + 2][O + 2], float d0[][N + 2][O +
      2], float u[][N + 2][O + 2], float v[][N + 2][O + 2], float w[][N + 2][O +
      2], float dt, float dtx, float dty, float dtz)
2 {
3   int i = blockIdx.x*blockDim.x + threadIdx.x;
4   int j = blockIdx.y*blockDim.y + threadIdx.y;
5   int k = blockIdx.z*blockDim.z + threadIdx.z;
6
7   int i0, j0, k0, i1, j1, k1;
8   float x, y, z, s0, t0, s1, t1, u1, u0;
9
10  if ((i >= 1) && (j >= 1) && (k >= 1) && (i <= M) && (j <= N) && (k <= O))
11  {
12
13    x = i - dtx * u[i][j][k]; y = j - dty * v[i][j][k]; z = k - dtz * w[i][j][
        k];
14    if (x < 0.5) x = 0.5; if (x > M + 0.5) x = M + 0.5; i0 = (int)x; i1 = i0 +
        1;
15    if (y < 0.5) y = 0.5; if (y > N + 0.5) y = N + 0.5; j0 = (int)y; j1 = j0 +
        1;
16    if (z < 0.5) z = 0.5; if (z > O + 0.5) z = O + 0.5; k0 = (int)z; k1 = k0 +
        1;
17
18    s1 = x - i0; s0 = 1 - s1; t1 = y - j0; t0 = 1 - t1; u1 = z - k0; u0 = 1 -
        u1;
19    d[i][j][k] = s0 * (t0*u0*d0[i0][j0][k0] + t1 * u0*d0[i0][j1][k0] + t0 * u1
        *d0[i0][j0][k1] + t1 * u1*d0[i0][j1][k1]) +
20      s1 * (t0*u0*d0[i1][j0][k0] + t1 * u0*d0[i1][j1][k0] + t0 * u1*d0[i1][j0
          ][k1] + t1 * u1*d0[i1][j1][k1]);
21  }
22 }
```

Listing 7.3. CUDA parallel kernel version for advection.

Note that the cited triple for loop in Listing 7.1 has been parallelized through (i,j,k) indices that point to the grid block threads (lines 3-5) in Listing 7.3. The protective line 10 in Listing 7.3 avoids exceeding the thread access outside the grid limits due to the distribution performed in Equation 7.10.

Essentially, the parallelization of Listing 7.1 in Listings 7.2 and 7.3 achieves the multiple calculation of advection by swapping the cells of the grid tunnel one at a time. The same calculation and CUDA C++ code is performed for the other functions in Algorithm 7.1, i.e., diffuse and project.

7.1.3 Jacobi relaxation with CUDA device pointers

Once of the problems arising in the diffuse and project functions is the feedback in the linear solver, which implements the discretization of Equation 7.9 with the following Poisson equation for each grid cell:

$$D^{n}_{i,j,k} = D^{n+1}_{i,j,k} - \frac{kdt}{h^3}\left(D^{n+1}_{i-1,j,k} + D^{n+1}_{i,j-1,k} + D^{n+1}_{i,j,k-1}, D^{n+1}_{i+1,j,k} + D^{n+1}_{i,j+1,k} + D^{n+1}_{i,j,k+1} - 6D^{n+1}_{i,j,k}\right) \tag{7.11}$$

where D^n and D^{n+1} are the grid cell velocity before and after diffusion, respectively; i,j,k is the spatial position in the grid; dt is the simulation time step value; h^3 is the volume of the cell; and k is the diffusion rate.

The previous equation must be rewritten in the form of Equation 7.12 to obtain the value $D^{n+1}_{i,j,k}$:

$$D^{n+1}_{i,j,k} = \frac{D^{n}_{i,j,k} + \frac{kdt}{h^3}\left(D^{n+1}_{i-1,j,k} + D^{n+1}_{i,j-1,k} + D^{n+1}_{i,j,k-1}, D^{n+1}_{i+1,j,k} + D^{n+1}_{i,j+1,k} + D^{n+1}_{i,j,k+1} + D^{n+1}_{i,j,k}\right)}{1 + \frac{kdt}{h^3}} \tag{7.12}$$

Equation 7.12 is the sparse linear solver system in the form $Ax = b$ [Ama09] in its serial version. This equation is solved with an iterative method for parallelization using Jacobi relaxation.

This linear solver feedback causes a race condition inside the CUDA kernel if no precautions are taken. Thus, the Jacobi approximation is performed in simulation to avoid this error as explained in Algorithm 7.2:

```
1  Function linearSolverKernel(x,x0,y,a,c)
2  |  i,j,k ← blockIdx_{x,y,z} × blockDim_{x,y,z} + threadIdx_{x,y,z}
3  |  if ((i,j,k ≥ 1)and(i,j,k ≤ Dim_{x,y,z})) then
4  |  |  y[i,j,k] ← x0[i,j,k] + a × (x[i−1,j,k] + x[i+1,j,k] + x[i,j−1,k] + x[i,j+1,k] +
   |  |    x[i,j,k−1] + x[i,j,k+1])/c
5  |  end
6
7  (...)
8
9  for t ← 1 < times for approximation do
10 |  linearSolverKernel <<< grid,block >>> (x,x0,y,a,x)
11 |  aux ← x
12 |  x ← y
13 |  y ← aux
14 end
```

Algorithm 7.2: Basic pseudocode of the linear solver with the Jacobi approximation.

Algorithm 7.2 is the solution to the race condition with good approximate convergence results.

Regarding the C++ CUDA implementation, Algorithm 7.2 is improved using a simple pointer swap that simulates lines 11-13 perfectly, as seen in the following listings:

```
1   // Linear solver
2
3 __global__ void kernelLinSolve(float x[][N + 2][O + 2], float x0[][N + 2][O +
      2], float y[][N + 2][O + 2], float a, float c)
4 {
5   int i = blockIdx.x*blockDim.x + threadIdx.x;
6   int j = blockIdx.y*blockDim.y + threadIdx.y;
7   int k = blockIdx.z*blockDim.z + threadIdx.z;
8
9   if ((i >= 1) && (j >= 1) && (k >= 1) && (i <= M) && (j <= N) && (k <= O))
10  {
11    // update for each cell
12    y[i][j][k] = (x0[i][j][k] + a * (x[i - 1][j][k] + x[i + 1][j][k] + x[i][j
          - 1][k] + x[i][j + 1][k] + x[i][j][k - 1] + x[i][j][k + 1])) / c;
13  }
14 }
```

Listing 7.4. The linear solver kernel explained in Algorithm 7.2.

```
1 void FluidCUDA::diffuse(float x[][N + 2][O + 2], float x0[][N + 2][O + 2],
     float diff, float dt, float dev_y[][N + 2][O + 2])
2 {
3   int max = MAX(MAX(M, N), MAX(N, O));
4   float a = dt * diff*max*max*max;
5
6   dim3 block(numBlocksX, numBlocksY, numBlocksZ);
7   dim3 thread(THREADS_X, THREADS_Y, THREADS_Z);
8
9   for (int k = 0; k < LINEARSOLVERTIMES; k++)
10  {
11    kernelLinSolve<<<block, thread>>>(x, x0, dev_y, a, 1.0f + 6.0f * a);
12    SWAP(x, dev_y); // Main trick
13  }
14 }
```

Listing 7.5. The trick that simulates lines 11-13 of Algorithm 7.2.

Let us take the previously allocated device **devU* and **devUPrev* arrays in Listing 7.5 as a illustrative example for the pointer swap trick. Figure 7.2 shows the initial pointer assignation; then, the pointers are swapped in line 9 of Listing 7.6 as seen in Figure 7.3. When the diffuse function is called, the pointers enter into a finite loop, swapping their pointers LINEARSOLVERTIMES iterations as indicated in Figures 7.4 and 7.5. After the diffusion kernel function call, the pointers remain in the stage of Figure 7.5 since they were passed by value.

```
1  // Simulation step
2
3  void FluidCUDA::velStep()
4  {
5     dim3 block(numBlocksX, numBlocksY, numBlocksZ);
6     dim3 thread(THREADS_X, THREADS_Y, THREADS_Z);
7
8     addSource<<<block, thread>>>(devU, devUPrev, dt); addSource<<<block,
          thread>>>(devV, devVPrev, dt); addSource<<<block, thread>>>(devW,
          devWPrev, dt);
9     SWAP(devUPrev, devU); // Sentence under study
10    diffuse(devU, devUPrev, visc, dt, devUy)); // Sentence under study
11    SWAP(devVPrev, devV);
12    diffuse(devV, devVPrev, visc, dt, devVy);
13    SWAP(devWPrev, devW);
14    diffuse(devW, devWPrev, visc, dt, devWy);
15    project(devU, devV, devW, devUPrev, devVPrev, devUPrevy);
16    SWAP(devUPrev, devU); SWAP(devVPrev, devV); SWAP(devWPrev, devW);
17    advect(devU, devUPrev, devUPrev, devVPrev, devWPrev, dt); advect(devV,
          devVPrev, devUPrev, devVPrev, devWPrev, dt); advect(devW, devWPrev,
          devUPrev, devVPrev, devWPrev, dt);
18    project(devU, devV, devW, devUPrev, devVPrev, devUPrevy);
19 }
```

Listing 7.6. The client function.

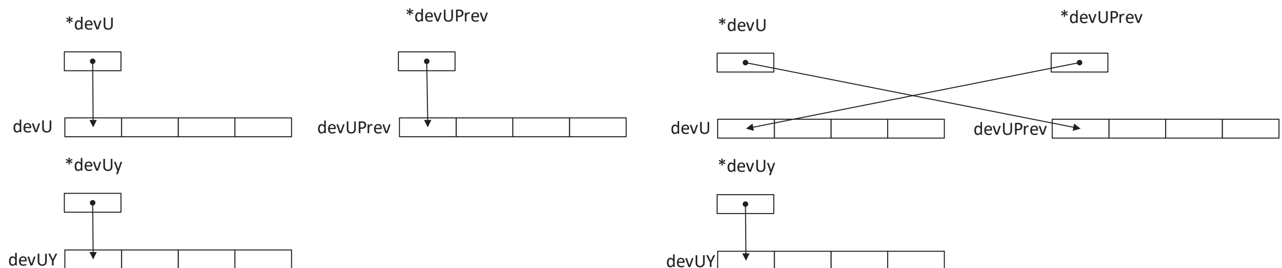


Figure 7.2. First step.

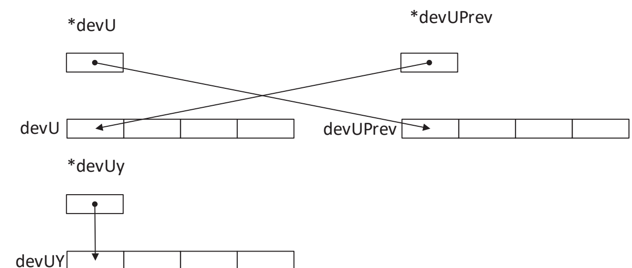


Figure 7.3. Second step.

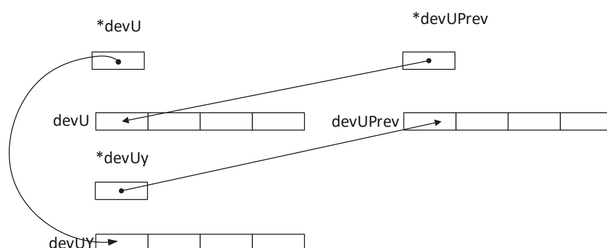


Figure 7.4. Third step.

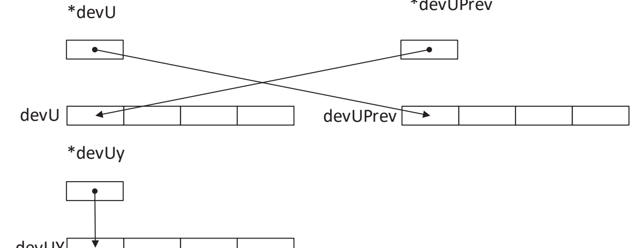


Figure 7.5. Fourth and final step.

7.2 Guide points and deformation

7.2.1 Guide points

Since the previously explained fluid class engine does not use density values, it takes advantage of force components to move the cumulus cloud in the wind direction. Each cloud entity has a pivot or guide point on a selected pseudosphere that is on the pole opposite to the wind origin as observed in Figure 7.6. This method retrieves the values of the UVW force components in Algorithm 7.1 to generate an incremental variable for this guide point according the status of the fluid class engine [JG19].

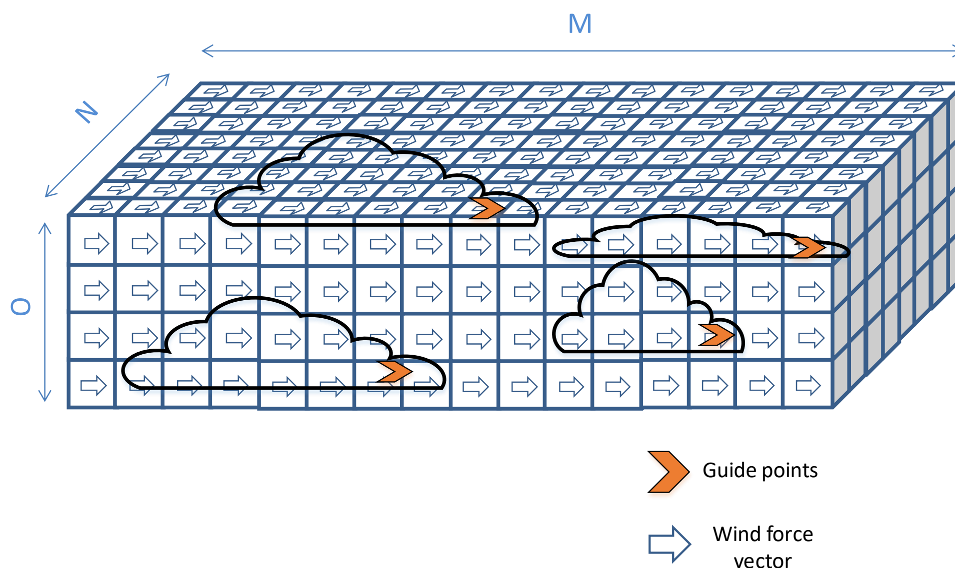


Figure 7.6. Four cumulus clouds with guide points drifting with the generated wind inside the whole fluid grid tunnel.

The grid used in the tests of the research has dimensions of $M = 100, N = 40, O = 40$. The grid can be seen as a wind tunnel where the force components are injected into $100 \times 40 \times 40 = 160000$ cells; however, when CUDA is not enabled, the CPU serial version should iterate the cells but with an increment to step over some rows or columns with the aim of achieving better performance.

The guide point is applied to the sphere that is farthest from the origin of the wind for better simulation and control. This guide point is fed with the R^3 force components of the fluid engine as specified in Equation 7.13:

$$sphPos(x,y,z)_i = \sum_{i=1}^{S} F_{U,V,W}(x,y,z) \tag{7.13}$$

where S is the number of pseudospheres and F is the wind force component in R^3 for the guide point. As an example, Figures 7.7 to 7.14 show the evolution of a cumulus cloud at different time points of its trajectory for the wind coming from the east.

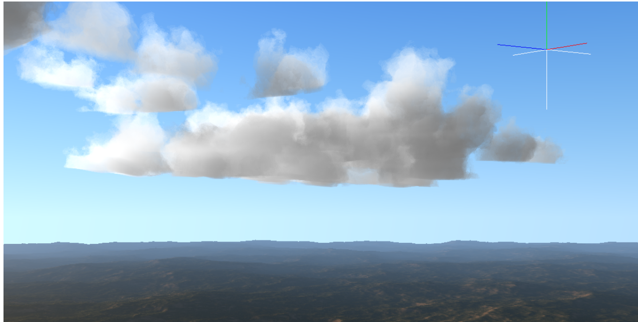


Figure 7.7. First frame.

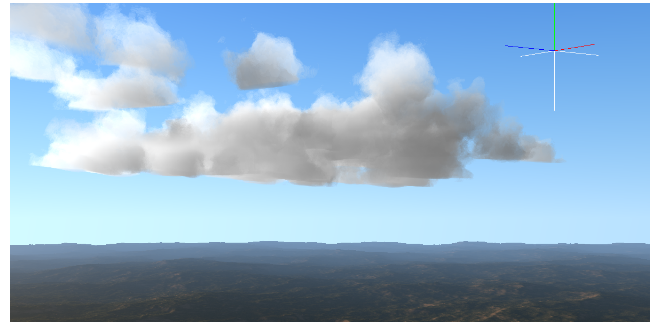


Figure 7.8. Second frame.

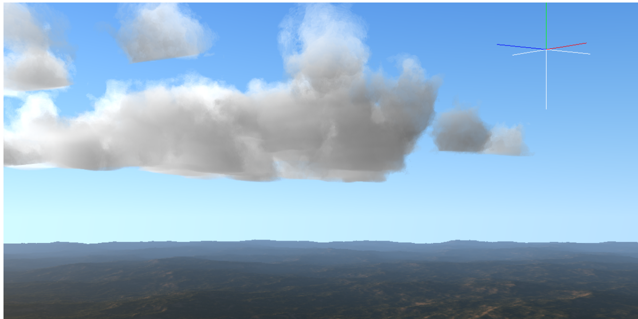


Figure 7.9. Third frame.

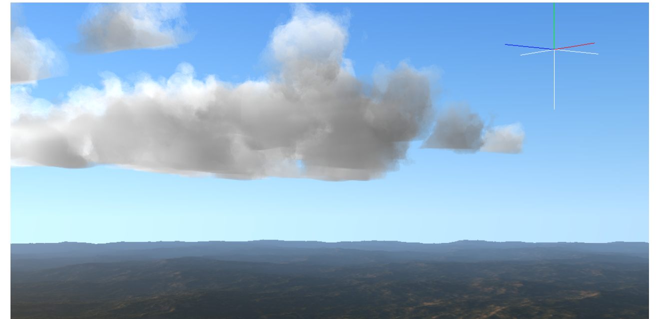


Figure 7.10. Fourth frame.

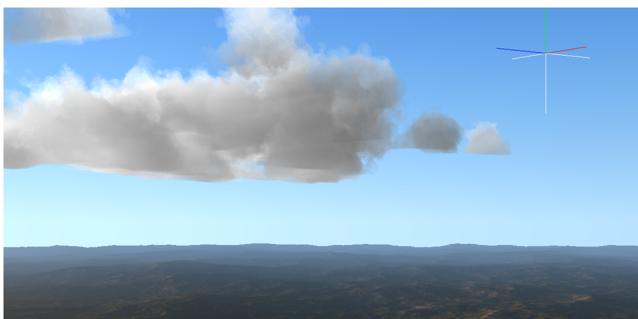


Figure 7.11. Fifth frame.

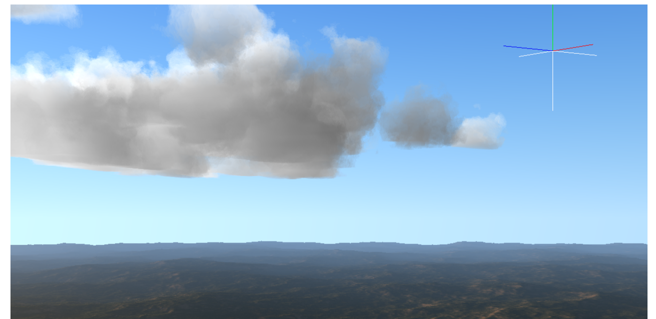


Figure 7.12. Sixth frame.

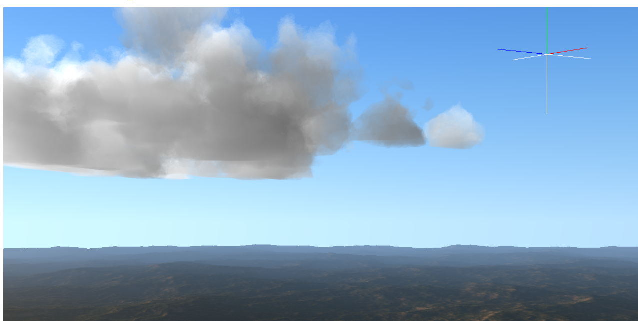


Figure 7.13. Seventh frame.

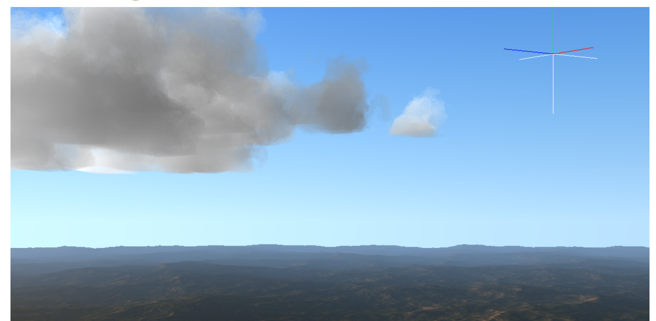


Figure 7.14. Eighth frame.

Other works, such as [Dob+00], have proposed an efficient and realistic method of cloud motion based on cellular automation. Their OpenGL implementation requires a small amount of computation and has low memory requirements as a result of the Boolean operations. Their research includes shadows of clouds and shafts of light.

7.2.2 Deformation

Since the assumed approach is ontogenetic, the cloud edge deformation is independent of the state of the fluid engine. Thus, the choice taken for modelling this feature is based on a modification of the WebGL work of [Qui15] but applied the efficient volumetric cloud rendering based on pseudospheres and fluid translation dynamics of this thesis. The model essentially consists of the increment of the fBm noise that is affected by a frame iterator and the wind direction as seen in Algorithm 7.3:

```
1  Function fBm(q⃗, wind⃗, frame)
2      q⃗ ← q⃗ + wind⃗ · frame
3      f ← 0
4      a ← 0.5
5      for i ← 0 < 5 do
6          f = Σ_{j=0}^{i} a · uniform(q⃗)
7          q⃗ = Π_{j=0}^{i} ⊗2.0
8          q⃗ = Σ_{j=0}^{i} 0.03
9          a = Π_{j=0}^{i} 0.5
10     end
11 return f
```

Algorithm 7.3: fBm function modification for cloud deformation.

The resulting deformation can be observed between Figure 7.7 and Figure 7.14, featuring very good results, especially in the area of fluid class engine adaptation.

7.3 Cloud morphing

In addition to the cloud motion generated by an efficient fluid simulator, many users in the computer game and visual arts fields are increasingly requiring new attractive special effects (FX) in their virtual outdoor products. Recently, researchers have begun to incorporate this requirement in their landscape generation software. The work of [YW11] is a notable example of a framework implementation for cloud modelling, rendering and morphing.

This thesis also researches the incorporation of some types of FX, especially with the morphing effect based on the work of [BN92] and [LWS98].

7.3.1 Morphing background

The alteration of an object shape into a different one is called *morphing*, which comes from the word «metamorphosis». As explained in [HB04], given two keyframes with different line segment numbers that represent the object transformation, we can adjust the object specification in one of the keyframes so that the number of edges or vertices are the same in the two keyframes as seen in Figure 7.15.

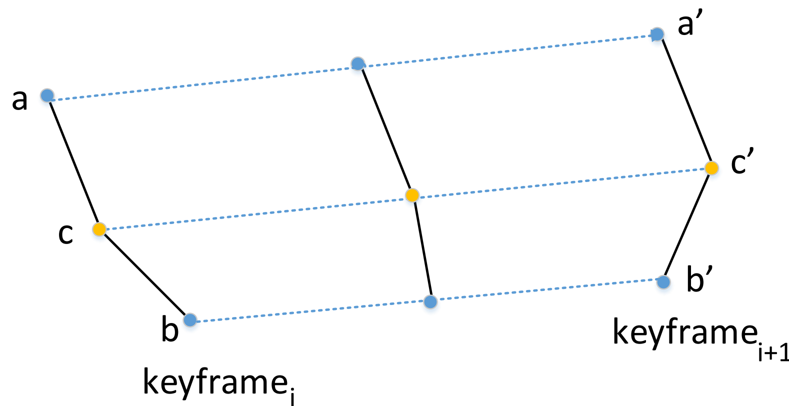


Figure 7.15. Since the the keyframe $i+1$ has an additional vertex, we add one vertex c between vertex a and b in keyframe i. By using linear interpolation to generate intermediate frames, a transition from c to c' is made.

As a consequence, the general rules to equalize the keyframes are the following:

- **Using edges:** Let E_i and E_{i+1} be the edges of a polygon in two consecutive keyframes.

$$E_{max} = max\left(E_i, E_{i+1}\right)$$
$$E_{min} = min\left(E_i, E_{i+1}\right)$$

Next, we calculate the following values:

$$T_e = E_{max} \text{ mod } E_{min}$$
$$T_s = \left\lfloor \frac{E_{max}}{E_{min}} \right\rfloor$$

Therefore, the preprocessing stages for the edge equalization are performed as follows:

$$Stage_1 = \frac{(T_e)_{Frame_{min}}}{T_s + 1}$$
$$Stage_2 = \frac{\text{rest of lines}_{Frame_{min}}}{T_s}$$

- **Using vertices:** Let V_i and V_{i+1} be vertices of a polygon in two consecutive keyframes.

$$E_{max} = max\left(V_i, V_{i+1}\right)$$
$$E_{min} = min\left(V_i, V_{i+1}\right)$$

Next, we calculate the following values:

$$T_{ls} = \left(V_{max} - 1\right) \text{ mod } \left(V_{min} - 1\right)$$
$$T_p = \left\lfloor \frac{V_{max} - 1}{V_{min} - 1} \right\rfloor$$

Therefore, the preprocessing stages for vertex equalization are performed as follows:

1. Add T_p points to T_{ls} line sections in $keyframe_{min}$.
2. Add $T_p - 1$ points to the rest of the edges in $keyframe_{min}$.

7.3.2 Proposed model

In a similar way as that explained in Section 7.3.1 for the vertices case, this thesis proposes a new approach for the transformation of two 3D wireframe meshes by moving each pseudoellipsoid barycenter (i.e., the vertex) of the source shape to the target shape through linear interpolation as described the following equation in GLSL, as cited in [JG19]:

$$f(x,y,a) = x \cdot (1 - a) + y \cdot a \qquad (7.14)$$

where two situations may arise: [1]

A) **Barycenters in the source > Barycenters in the target mesh:** In this case, we assign a direct correspondence between the barycenters in the source and target in iterative order. The excess barycenters in the source are randomly distributed by overlapping them across the target barycenters with the calculation of the modulus between their barycenters as seen in Figure 7.16.

B) **Barycenters in the target > Barycenters in the source mesh:** The opposite operation implies a random reselection of the excess source barycenters to duplicate them and generate a new interpolation motion to the target mesh as seen in Figure 7.17.

[1] Although just one of the choices below is required with the inversion of the a variable from 1 to 0, an alternative approach is proposed here.

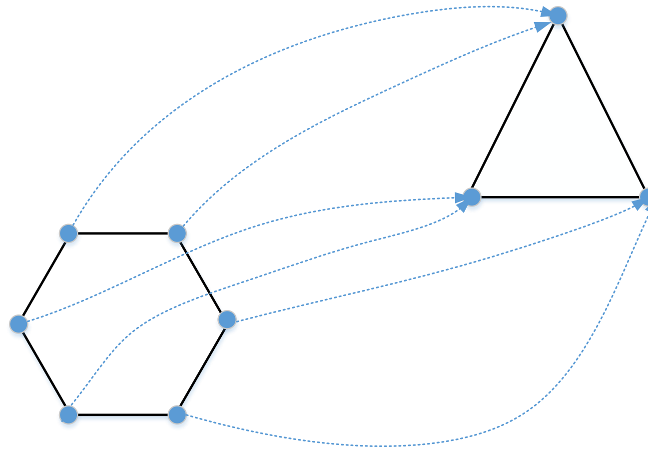


Figure 7.16. Case A. The excess barycenters in the hexagon are randomly distributed and overlapped over the triangle barycenters.

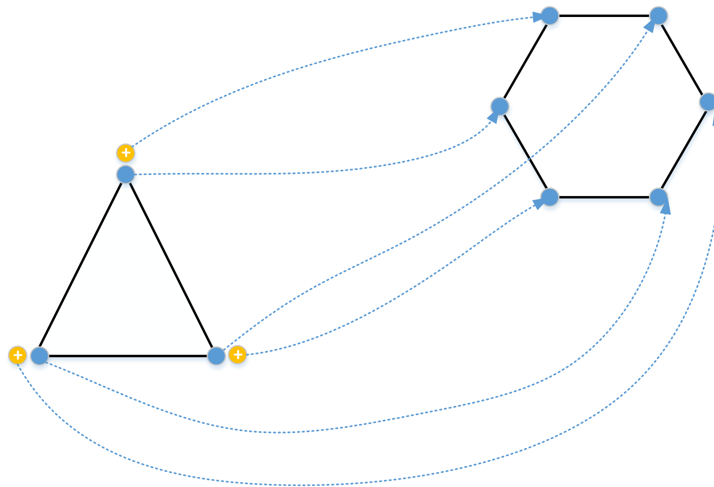


Figure 7.17. Case B. The required barycenters are added to the triangle for a random reselection to the hexagon barycenters.

Another problem that arises in the morphing model is the orientation or correct rotation of the pseudoellipsoids when they are moving from the source to the target. Thus, the first difficulty is to relocate the source ellipsoid in the direction of the triangle. This drawback is successfully overcome as explained in Section 6.5. Another difficulty is the incremental rotation when the source ellipsoid travels linearly to the destination with the objective to achieve smooth rendering during the animation. Figure 7.18 illustrates this issue.

As explained in [JG18], the first rotation is generated with Rodrigues' transformation formulas (Equations 7.15 and 7.16) given the first angle θ.

$$R_1 = e^{A\theta} = I + sin(\theta)A + (1 - cos(\theta))A^2 \quad (7.15)$$

$$R_2 = e^{A\alpha_i} = I + sin(\alpha_i)A + (1 - cos(\alpha_i))A^2 \quad (7.16)$$

where α_i is the incremental angle from 0 to α_n, which is the final angle in the target pseudoellipsoid orientation.

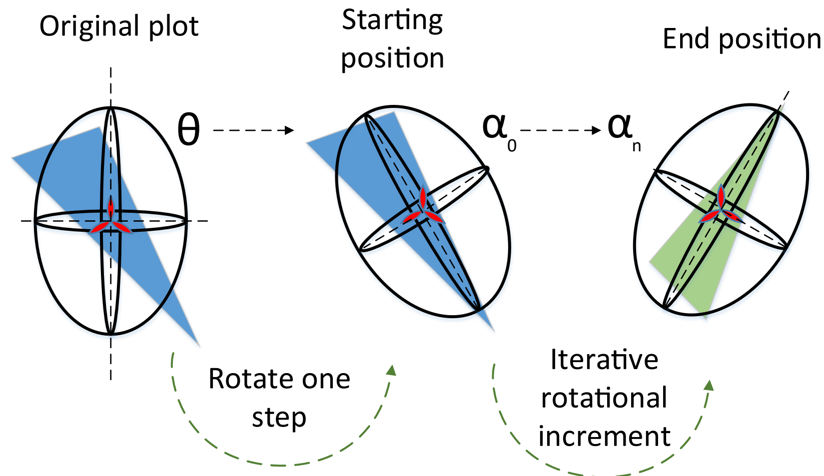


Figure 7.18. Example layout of the rotation process for 3D meshes.

Equation 7.15 is performed once in the CPU, whereas Equation 7.16 is calculated in each morphing iteration until the final angle is reached. Usually, the morph animation ends when all the pseudoellipsoids have reached the last rotation angle.

Therefore, the GLSL shader performs the transformation of Equation 7.17 during the rendering.

$$T_{\text{final}} = R_2 \cdot R_1 \cdot \text{Point of the pseudoellipsoid} \tag{7.17}$$

An additional difficulty found during the proposed morphing model is the definition of the bounding boxes. Due to the complexity of the animation control, the implementation uses a shared bounding box with the size of both meshes. On the other hand, an efficiency improvement was achieved with shadowing as a result of the linear interpolation of the precomputed light voxels.

The mesh deformation is also applied during the metamorphosis by using the method explained in Section 7.2.2 but without the fluid dynamics engine.

Finally, Figures 7.19 to 7.26 show an example of the progression of a hand-to-rabbit mesh metamorphosis.

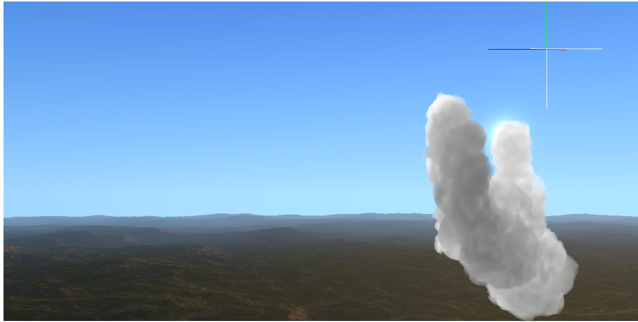


Figure 7.19. First step.

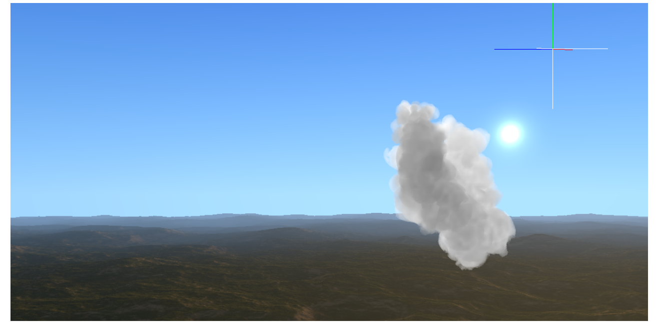


Figure 7.20. Second step.

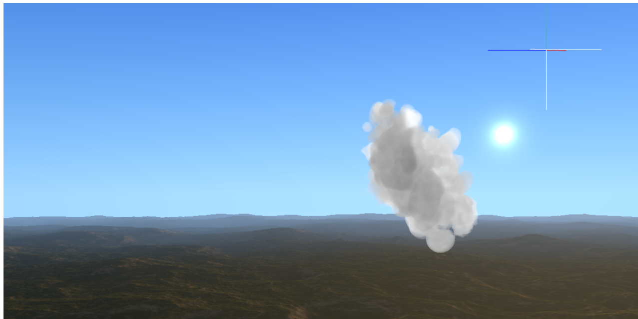


Figure 7.21. Third step.

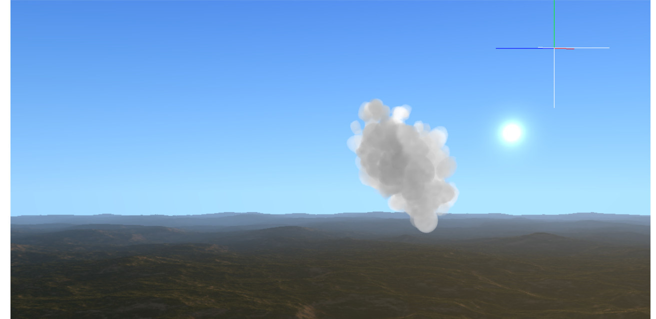


Figure 7.22. Fourth step.

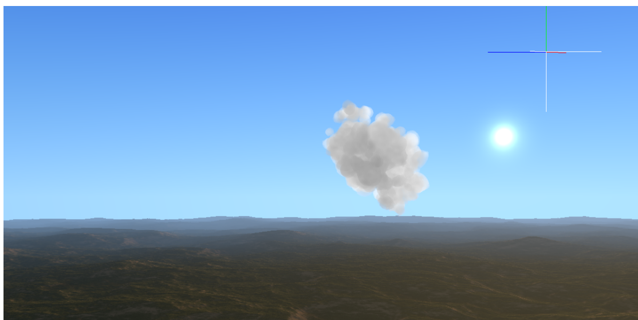


Figure 7.23. Fifth step.

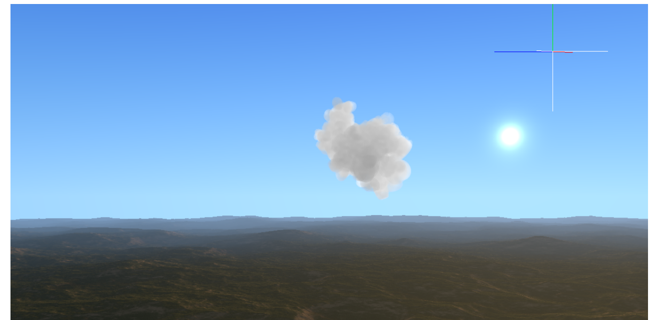


Figure 7.24. Sixth step.

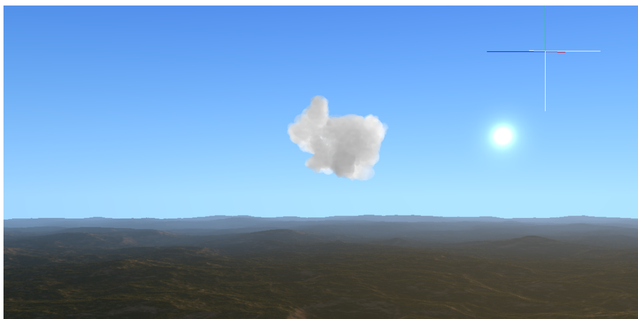


Figure 7.25. Seventh step.

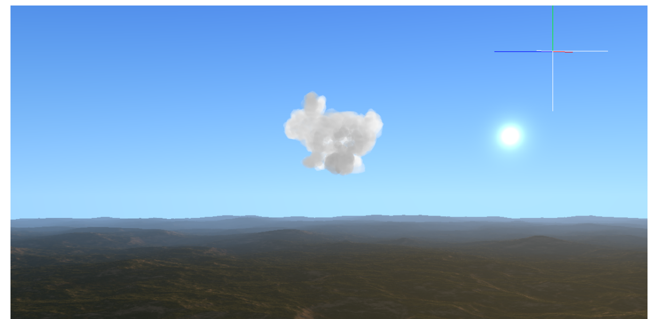


Figure 7.26. Eighth step.

7.4 Summary

We have reviewed in the present chapter the main concepts related to fluid simulation for advection and convection in the computer graphics field. The explanation of the behaviour and internal methods of the fluid engine class is useful to understand the cloud dynamics research work. This chapter also introduces the critical parts of the CUDA code parallelization to achieve a more efficient and effective C++ implementation. It also illustrates the cloud deformation principles applied in this thesis and the novel technique of the guide points. Finally, an exposition is included of the essential ideas to perform 3D cloud model morphing along with a background introduction to this effect.

CHAPTER 8

Results and benchmarks

To confirm the initial hypothesis, benchmarks and measurements were made using different GPUs to evaluate that the developed algorithms yield good realism and performance in all kind of graphic cards. For this reason, this chapter begins with a deep complexity analysis of the essential algorithm implemented on the GPU, which is performed in Section 8.1. A complete collection of performance metric graphs in both serial and parallel execution tests is also analysed in Section 8.2 for the static and dynamic cloud rendering models. Section 8.3 explains the impact of CUDA parallelism in the overall system speedup, and finally, Section 8.4 deploys a wide range of cloud taxonomy figures to assess the quality of the rendering.

8.1 Complexity analysis

Analytical verification of Algorithm 4.1 that runs on the GPU has been undertaken to assess its complexity. The following analysis has been developed using [BB96] notation.

Considering $|B|$ as the number of collisions with clouds per ray, n as their overlapped bounding boxes near-plane resolution area in pixels, s as the number of pseudospheroids in their bounding boxes, c as the size of selected candidates to be rendered, and d as the depth of raymarching, Algorithm 4.1 has the following execution times: [1]

- If we assume no sphere collisions, the Insertion Sort algorithm will not have any elements to classify in the *best-best case*, so we will obtain an asymptotic execution time of:

$$t_{best-best} = \sum_{i=1}^{|B|} \sum_{j=1}^{n} \left(1 + \underbrace{\sum_{t=1}^{s} (1+6)}_{getCandidates()} \right) \simeq \Omega(|B|ns) \tag{8.1}$$

[1] The numeric constants refer to the instruction count.

- In the *worst-best case* the candidates list is already sorted, so no swap will be done.

$$t_{worst-best} = \sum_{i=1}^{|B|}\sum_{j=1}^{n}\left(1+\underbrace{\sum_{t=1}^{s}(1+6+5)}_{getCandidates()}\right)+\sum_{j=1}^{n}\left(\underbrace{c}_{sortCandidates()}+1+\underbrace{\sum_{w=1}^{c}\left(1+2+\sum_{z=1}^{d}20\right)}_{\text{Raymarching iterations}}\right) \simeq o-\Omega\Big(n(4c+20dc+12|B|s)\Big) \quad (8.2)$$

- The *worst-worst case* happens when a complete collision is detected and the candidate list is in descending order, so the resulting asymptotic worst-case execution time is:

$$t_{worst-worst} = \sum_{i=1}^{|B|}\sum_{j=1}^{n}\left(1+\underbrace{\sum_{t=1}^{s}(1+6+5)}_{getCandidates()}\right)+\sum_{j=1}^{n}\left(\underbrace{c^2}_{sortCandidates()}+1+\underbrace{\sum_{w=1}^{c}\left(1+2+\sum_{z=1}^{d}20\right)}_{\text{Raymarching iterations}}\right) \simeq o-o\Big(n(c^2+3c+20dc+12|B|s)\Big) \quad (8.3)$$

- Regarding *the average case* we assume a *p* probability of sphere collision, so an approximate measure of total collisions will be calculated as a percentage of total spheres in scene.

$$t_{avg} = \sum_{i=1}^{|B|}\sum_{j=1}^{n}\left(1+\underbrace{\sum_{t=1}^{s}(1+6+5)}_{getCandidates()}\right)+\sum_{j=1}^{n}\left(\underbrace{c^2}_{sortCandidates()}+1+\underbrace{\sum_{w=1}^{s\cdot p}\left(1+2+\sum_{z=1}^{d}20\right)}_{\text{Iterations probability}}\right) \simeq o\Big(n(c^2+20dsp+12|B|s+3ps)\Big) \quad (8.4)$$

The analysis indicates that, despite the algorithm having quadratic complexity in the worst case for a single-frame buffer pixel, modelling the cumulus with fewer spheres reduces the number of hits, resulting in faster sorting and execution.

Some tests were performed by replacing the *Insertion Sort* algorithm with *iterative Quicksort* which contains $O(nlog_2n)$ instead of $O(n^2)$, expecting that it would provide better performance. However, no improvement in FPS was obtained in these experiments. On the contrary, there was a decrease in the overall frames-per-second. The cause of this reduction is an increase in memory access on the GPU caused by the Quicksort algorithm, so in spite of

the in-line optimizations it could not be exceeded the speed of *Insertion Sort*. Another alternative for the *Insertion Sort algorithm* is *Batcher's Odd–Even Mergesort* as cited by Kipfer and Westermann [MR05] due to the better low-level parallelization and its worst-case complexity parallel time, represented by $O(log^2(n))$.

8.2 Benchmark tests

8.2.1 Static rendering

A set of tests was performed on the algorithm suite with using an nVidia GeForce 8800 GTS (96 cores), a GeForce 1030 GT (Pascal, 384 cores), a GeForce GTX 1050 non-Ti (Pascal, 640 cores) and a GeForce GTX 970 (1664 cores), running on a 64-bit Core i7 CPU 860@2.80 GHz (first generation, 2009) with 6 GB random access memory (RAM). The project was implemented entirely in C++ using the OpenGL and GLM math libraries for the host side and the OpenGL Shading Language for the GPU side. The raymarching step size was determined by λ in line 48 of Algorithm 4.1 for the cumulus test and was constant at 0.1 for the 3D mesh tests. Promising results were obtained when the cloud was far from the camera with the 8800 GTS at 800×600 and 640×480 pixels, especially with metaballs. The same algorithm suite performs perfectly in all resolutions using the nVidia GTX 970, GT 1030, and GTX 1050 non-Ti, in particular when rendering clouds derived from 3D meshes. Besides this, a significant $2\times$ frame rate improvement in the nVidia 8800 GTS was achieved in relation to [Bou+08] for cumulus rendering. For this benchmark, we used 35 spheres for cumulus generation and the R rotation for the 3D rabbit mesh. Both tests used a grid size of $20 \times 20 \times 20$ voxels and a uniform hypertexture of 64^3 single-precision floats. The pre-computation in the CPU took under 0.010 s thanks to the no duplicate tracing algorithm. Without the NDT algorithm, the CPU execution time would double for the aforementioned cumulus in all cases. The graphic driver used the factory default configuration in all tests. The CPU load did not exceed 15% and and host memory usage did not exceed 24.5 MB in all tests when running at a resolution of 1920×1080 pixels. Regarding the GPU hardware usage, with the nVidia 1030 GT, for instance, the top mark frame-buffer usage was 21% at maximum frames-per-second, the bus interface was 4% at maximum power, and the maximum memory allocation peak was up to 7%.

Card/OpenGL Euclidean Distance	**Cumulus**	**3D Model**
GT 1030	$32 \to \infty$	$25 \to \infty$
GTX 1050 non-Ti	$24 \to \infty$	$20 \to \infty$
GTX 970	$0 \to \infty$	$12 \to \infty$

Table 8.1. The table above shows the minimum distance from the cloud at which Full High-Definition (HD) 1920 × 1080 pixels rendering reaches 30 FPS (minimum real-time). This distance is suitable for scenarios where getting close to the surface of the cloud is required.

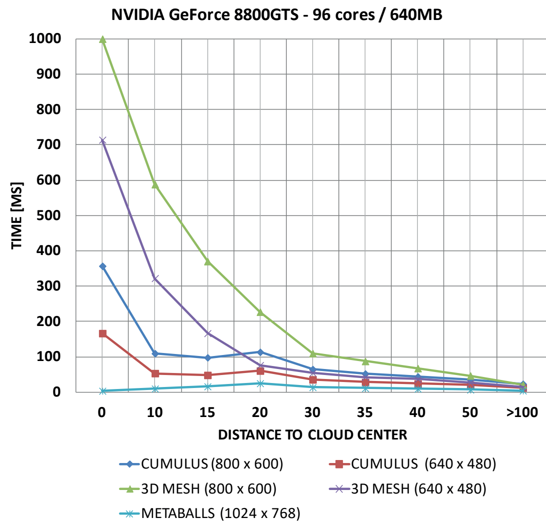


Figure 8.1. With the GeForce 8800 GTS the performance at 800 × 600 pixels overcomes the limit of the hyperrealistic method shown in [Bou+08].

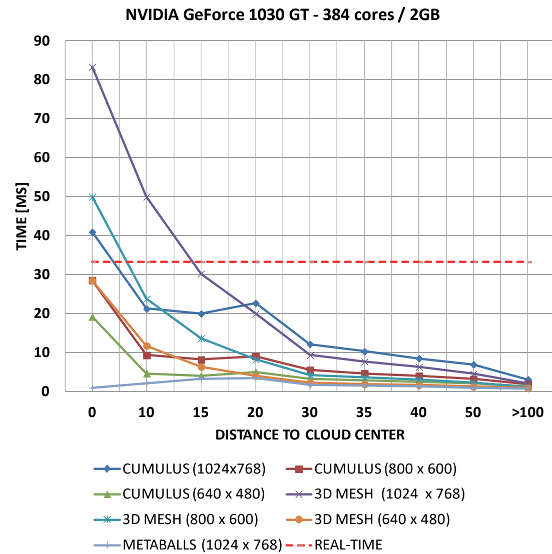


Figure 8.2. With the GeForce 1030 GT the performance is optimum in most cases, except when the level of detail (LOD) equation is manually bypassed to force higher quality.

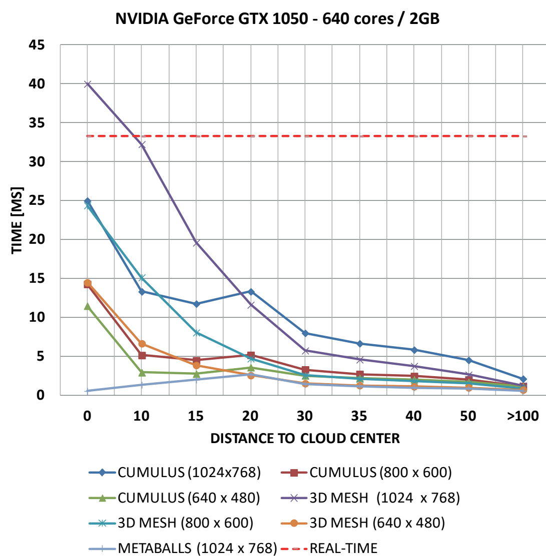


Figure 8.3. Performance in a GeForce GTX 1050 non-Ti is optimum in 99% of cases.

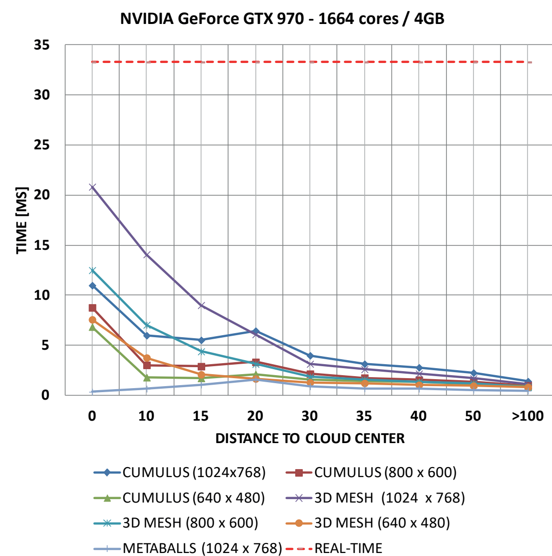


Figure 8.4. Based on empirical tests in a GeForce GTX 970, the proposed model achieves a geometric frame rate increment in all algorithms. Results are very promising for the both cumulus and 3D mesh tracing algorithms in all resolutions, including Full-HD.

8.2.2 Dynamic rendering and morphing

For the dynamic cloud rendering and the morphing effect, a set of benchmark tests were performed using an nVidia GeForce 1030 GT (Pascal, 384 cores), a GeForce GTX 1050 non-Ti (Pascal, 640 cores) and a GeForce GTX 1070 non-Ti (Pascal, 1920 cores), running on a 64-bit Core i7 CPU 860@2.80 GHz (first generation, 2009) with 6 GB RAM. As seen in Section 8.2.1 the project is implemented in C++ using OpenGL and GLM libraries with GLSL for the shader language. The Visual C++ 2017 compiler was set to **O2**(maximum speed optimization) and **Ot** (favouring optimization for speed over optimization for size). In addition, the **Qpar** options were enabled to generate CPU parallel code. In the CUDA compiler side, the **compute_61** flag was set for device code generation, and the **O2** option to maximize speed and the **fast math** operations flag were enabled. As in the static benchmarks, the λ raymarching step was established in 0.1 for the morphing effect. The cumulus dynamic tests were performed over a moving sea scape with a real sky function using four clouds in the scene with 35 spheres per each one (140 spheres in total). The metaballs tests used a plain terrain landscape with two clouds with 6 spheres each one. With the aim to analyse the CUDA parallel performance, two benchmark versions were defined for each graphics card: a 10^3 precomputed light grid size with a $100 \times 20 \times 40$ fluid volume and a 40^3 precomputed light grid size with a $100 \times 40 \times 40$ fluid volume.

The following tables and graphs demonstrate the performance metrics for different graphics cards.

Card/OpenGL Euclidean Distance	Cumulus	3D Model
GTX 1070 non-Ti (CPU)	$35 \to \infty$	$24 \to \infty$
GTX 1070 non-Ti (CUDA)	$35 \to \infty$	

Table 8.2. Minimum distance from the cloud at which full HD (1920×1080) reaches 30 FPS in an nVidia GTX 1070 non-Ti with a problem size of 10^3 for the precomputed light grid and $100 \times 20 \times 40$ for the fluid engine grid.

Card/OpenGL Euclidean Distance	Cumulus
GTX 1070 non-Ti (CPU)	$40 \to \infty$
GTX 1070 non-Ti(CUDA)	$37 \to \infty$

Table 8.3. Minimum distance from the cloud at which full HD (1920×1080) reaches 30 FPS in an nVidia GTX 1070 non-Ti with a problem size of 40^3 for the precomputed light grid and $100 \times 40 \times 40$ for the fluid engine grid.

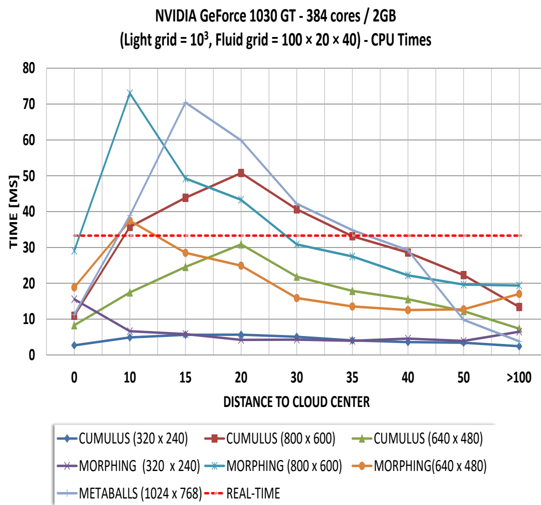


Figure 8.5. GeForce 1030 GT CPU simulation benchmark; 79.3% of the samples are above 30 FPS (see Table 8.4). The performance at 640×480 is good enough in this case.

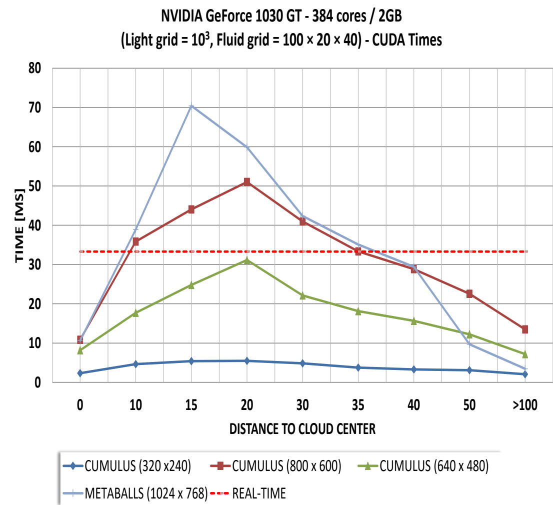


Figure 8.6. GeForce 1030 GT CUDA simulation benchmark; 75% of the samples in the graph are above 30 FPS (see Table 8.5). The parallel GPU overhead behaviour is very similar to that shown in Figure 8.5.

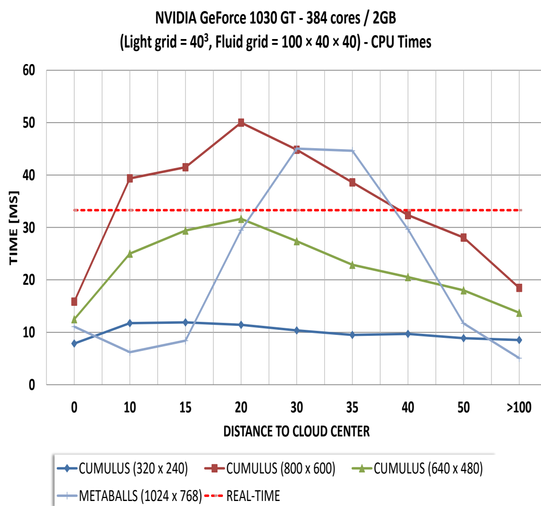


Figure 8.7. GeForce 1030 GT CPU simulation benchmark with a greater problem size; 80% of the samples fall over 30 FPS as seen in Table 8.6.

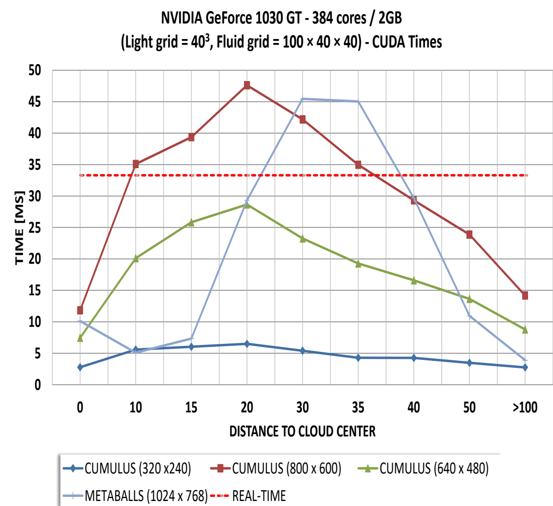


Figure 8.8. GeForce 1030 GT CUDA simulation benchmark with a greater problem size; 80% of the samples fall over 30 FPS as seen in Table 8.7. More overall speedup is gained.

	Dist.	CUMULUS (320 x 240)	CUMULUS (640 x 480)	CUMULUS (800 x 600)	MORPH. (320 x 240)	MORPH. (640 x 480)	MORPH. (800 x 600)	METABALLS (1024 x 768)
	0	374.1	120.8	91.4	64.3	53.1	34	88.8
	10	205	57.3	28	150.9	26.7	13.7	25.7
	15	178.5	40.7	22.8	171.3	35.1	20.3	14.2
	20	176.5	32.4	19.7	237.2	40.1	23.1	16.7
	30	197	45.8	24.6	233.2	63	32.4	23.7
	35	247.1	55.9	30.2	253.4	73.9	36.4	28.7
	40	278.2	64.3	35	218.6	79.8	45	34.2
	50	292.1	81.8	44.9	255.1	78.5	50.9	101.8
	>100	408.6	136.2	74.6	153.9	58.7	51.6	263.1
MEAN	25	261.9	41.2	63.52	193.1	56.54444444	34	66.32222222
STANDARD DEVIATION		84.56124999	35.96838118	25.14786826	63.31007029	19.37537813	13.4307483	80.29523301
TOTAL MEAN		59.26						

Table 8.4. FPS metrics for the GeForce GT 1030 in CPU simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size.

	Distance	CUMULUS (320 x 240)	CUMULUS (640 x 480)	CUMULUS (800 x 600)	METABALLS (1024 x 768)
	0	430.9	122.4	92.1	94.2
	10	216.2	56.5	27.9	25.7
	15	185	40.3	22.7	14.2
	20	182.7	32.1	19.6	16.7
	30	207.2	45.2	24.4	23.6
	35	265.9	55.1	30	28.5
	40	304.4	63.8	34.7	34
	50	324.7	81.7	44.3	103.4
	>100	485.3	139.2	74.1	289.3
MEAN	25	289.1444444	70.7	41.08888889	69.95555556
STANDARD DEVIATION		108.8698433	37.15360009	25.29414183	88.65798203
TOTAL MEAN		69.59514032			

Table 8.5. FPS measures for the GeForce GT 1030 in CUDA simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size.

	Distance	CUMULUS (320 x 240)	CUMULUS (640 x 480)	CUMULUS (800 x 600)	METABALLS (1024 x 768)
	0	127.4	80.1	63.1	90.3
	10	85.2	40	25.4	161.2
	15	84.1	34	24.1	118.9
	20	87.5	31.6	20	33.9
	30	96.6	36.5	22.3	22.2
	35	105.4	43.7	25.9	22.4
	40	103.3	48.7	30.9	33.7
	50	112.7	55.5	35.6	85.5
	>100	117.3	72.9	54.1	197.2
MEAN	25	102.1666667	49.2222222	33.4888889	85.0333333
STANDARD DEVIATION		15.20197356	17.2243126	15.12906511	63.91236187
TOTAL MEAN		44.72961195			

Table 8.6. FPS measures for the GeForce GT 1030 in CPU simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size.

	Distance	CUMULUS (320 x 240)	CUMULUS (640 x 480)	CUMULUS (800 x 600)	METABALLS (1024 x 768)
	0	360.1	134.1	84.6	98.9
	10	179.1	49.7	28.5	196.8
	15	165.8	38.7	25.4	136.5
	20	153.9	34.9	21	34.1
	30	185.6	43	23.7	22
	35	233.8	51.9	28.6	22.2
	40	234.7	60.3	34.1	33.7
	50	287.3	73.3	41.9	91.4
	>100	364.3	114	70.5	256.7
MEAN	25	240.5111111	66.65555556	39.81111111	99.14444444
STANDARD DEVIATION		80.43863258	34.87391974	22.52656457	83.81624412
TOTAL MEAN		65.06972514			

Table 8.7. FPS metrics for the GeForce GT 1030 in CUDA simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size.

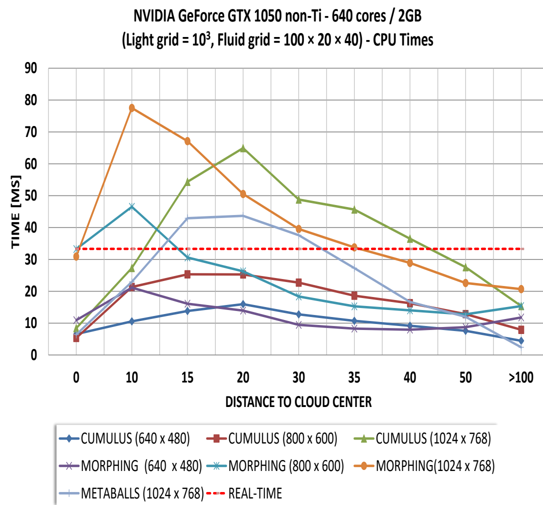


Figure 8.9. GeForce 1050 GTX CPU simulation benchmark; 77.7% of the samples are above 30 FPS (see Table 8.8). The performance at 800×600 is acceptable.

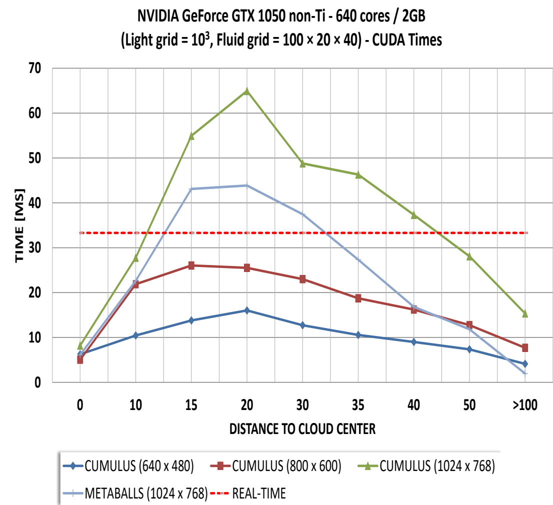


Figure 8.10. GeForce 1050 GTX CUDA simulation benchmark; 77.7% of the samples are above 30 FPS (see Table 8.9). The performance is slightly higher in this version.

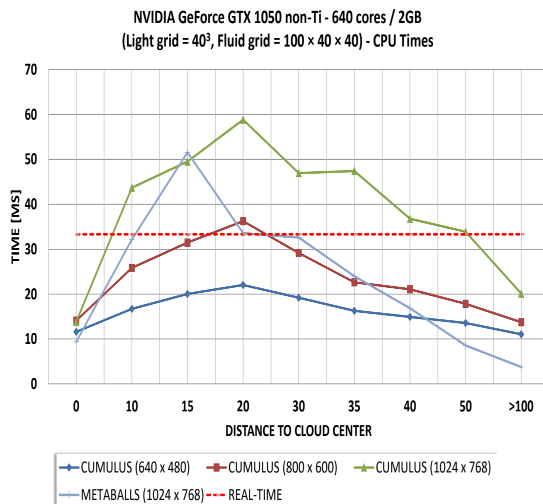


Figure 8.11. GeForce 1050 GTX CPU simulation benchmark with a greater problem size; 72.2% of the samples are above 30 FPS (see Table 8.10). The performance is slightly lower than that of its counterpart.

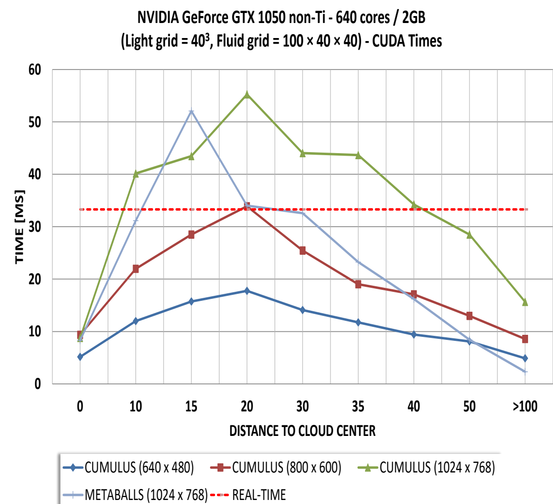


Figure 8.12. GeForce 1050 GTX CUDA simulation measures; 75% of the samples are above 30 FPS (see Table 8.11). The CUDA version is observed to improve the CPU times.

	Dist.	CUMULUS (640 x 480)	CUMULUS (800 x 600)	CUMULUS (1024 x 768)	MORPH. (640 x 480)	MORPH. (800 x 600)	MORPH. (1024 x 768)	METABALLS (1024 x 768)
	0	152.5	187.4	119.1	91.7	30	32.4	155.9
	10	94.9	46.9	36.6	47.3	21.5	12.9	43.6
	15	72.4	39.5	18.4	62.2	32.7	14.9	23.3
	20	62.7	39.6	15.4	71.8	38.1	19.8	22.9
	30	78.4	44	20.5	105.8	54.5	25.3	26.6
	35	93.4	53.8	21.9	120.9	65.5	29.6	36.7
	40	108.9	61.6	27.4	126	71.4	34.6	60
	50	132.2	78	36.3	114.7	78.3	44.3	83.6
	>100	224.7	126.3	65.2	85.1	65.1	48.4	415.4
MEAN	25	113.3444444	75.23333333	40.08888889	91.72222222	51	29.13333333	96.44444444
STANDARD DEVIATION		50.63213187	50.20829115	33.27042397	27.47297662	20.58749621	12.29796731	126.8660032
TOTAL MEAN		47.22630079						

Table 8.8. FPS for the GeForce GTX 1050 non-Ti in CPU simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size.

	Distance	CUMULUS (640 x 480)	CUMULUS (800 x 600)	CUMULUS (1024 x 768)	METABALLS (1024 x 768)
	0	159.7	198.6	122	163.9
	10	95.8	45.7	36	44.2
	15	72.5	38.4	18.2	23.2
	20	62.4	39.2	15.4	22.8
	30	78.6	43.5	20.5	26.7
	35	94.7	53.4	21.6	36.6
	40	111	61.7	26.8	59.4
	50	136	78.5	35.6	84.6
	>100	243.6	130.1	65.1	512.6
MEAN		117.1444444	76.56666667	40.13333333	108.2222222
STANDARD DEVIATION		56.60207839	54.10466708	34.21012277	158.099342
TOTAL MEAN		53.72849026			

Table 8.9. FPS for the GeForce GTX 1050 non-Ti in CUDA simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size.

	Distance	CUMULUS (640 x 480)	CUMULUS (800 x 600)	CUMULUS (1024 x 768)	METABALLS (1024 x 768)
	0	86.2	71.1	72.2	106.7
	10	59.9	38.7	22.9	31.1
	15	49.9	31.8	20.2	19.4
	20	45.4	27.6	17	29.8
	30	52.1	34.3	21.3	30.7
	35	61.4	44.2	21.1	41.7
	40	67.1	47.5	27.2	59.3
	50	73.7	56.1	29.5	116.6
	>100	90.5	72.8	49.7	266.26
MEAN	25	65.13333333	47.12222222	31.23333333	77.95111111
STANDARD DEVIATION		15.80023734	16.48785748	18.14414506	78.75352063
TOTAL MEAN		40.46738581			

Table 8.10. FPS metrics for the GeForce GTX 1050 non-Ti in CPU simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size.

	Distance	CUMULUS (640 x 480)	CUMULUS (800 x 600)	CUMULUS (1024 x 768)	METABALLS (1024 x 768)
	0	193.6	106.7	114.1	120.5
	10	83.4	45.5	24.9	32.1
	15	63.6	35.1	23	19.2
	20	56.3	29.5	18.1	29.4
	30	70.9	39.3	22.7	30.7
	35	85.2	52.6	22.9	43
	40	106.2	58.5	29.2	61.7
	50	123.4	77	35.1	118.5
	>100	205	116.6	63.9	430.2
MEAN	25	109.7333333	62.31111111	39.32222222	98.36666667
STANDARD DEVIATION		54.88098487	31.36755666	31.19866895	130.1020369
TOTAL MEAN		49.79777642			

Table 8.11. FPS benchmarks for the GeForce GTX 1050 non-Ti in CUDA simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size.

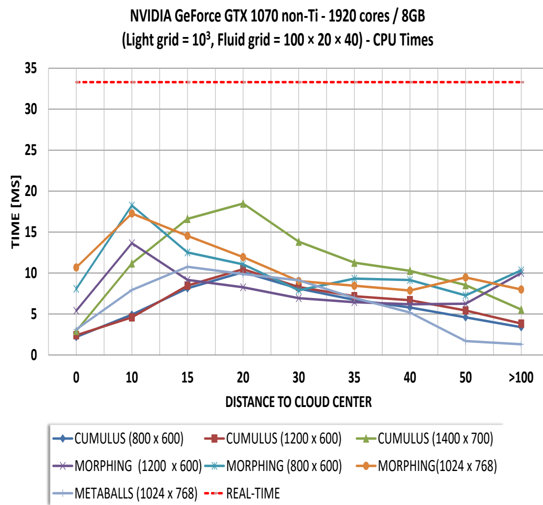


Figure 8.13. GeForce 1070 GTX CPU simulation benchmarks; 100% of the samples are above 30 FPS (see Table 8.12). The overall performance is optimal.

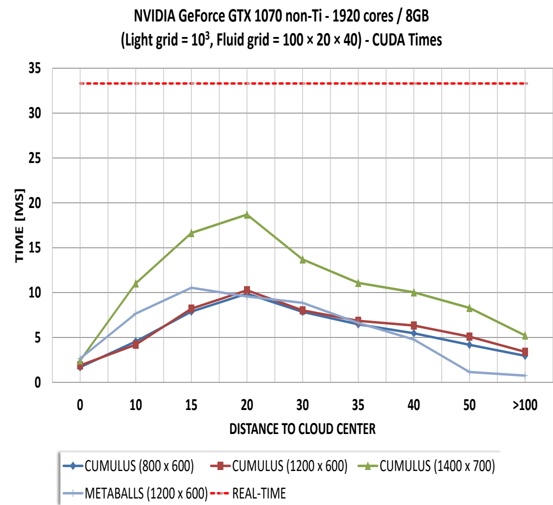


Figure 8.14. GeForce 1070 GTX CUDA simulation measures; 100% of the samples are above 30 FPS (see Table 8.13). The CUDA version improves the CPU times optimally.

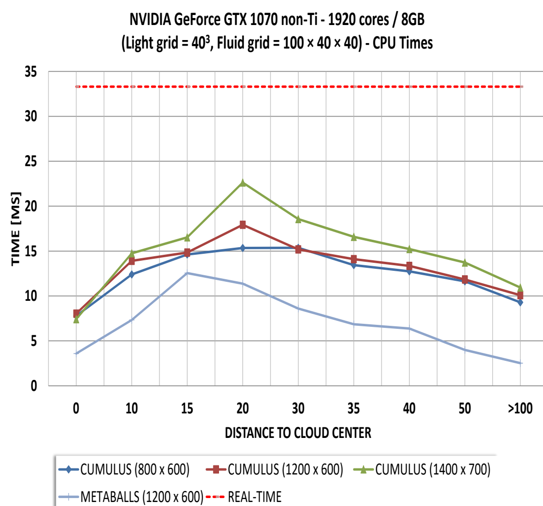


Figure 8.15. Performance of a GeForce GTX 1070 non-Ti with CPU simulations is optimal in 100% of the cases for a greater problem size as seen in Table 8.14.

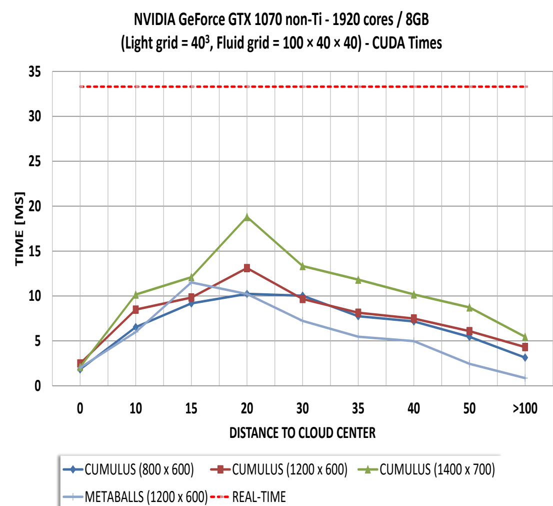


Figure 8.16. GeForce 1070 GTX CUDA simulation metrics; 100% of the samples are above 30 FPS (see Table 8.15). The CUDA version improves the CPU times optimally.

	Dist.	CUMULUS (800 x 600)	CUMULUS (1200 x 600)	CUMULUS (1400 x 700)	MORPH. (800 x 600)	MORPH. (1200 x 600)	MORPH. (1400 x 700)	METABALLS (1200 x 600)
	0	455.7	415.8	341.5	185	124	93.6	320.1
	10	203.2	217.9	89.5	73.3	54.8	57.9	126.2
	15	122.9	118.2	60.2	109.1	79.9	68.8	92.9
	20	98.9	95.5	54.1	121	90.4	83.7	101.3
	30	123.4	120.6	72.3	144.2	126.7	110.8	109.7
	35	148.5	139.5	88.7	155.1	107.2	118.3	143.6
	40	172.4	149.8	97.4	161.6	109.4	127.1	193.4
	50	217.9	184	117.2	160	137.6	105.6	587.9
	>100	294.2	260.8	180.3	100	96.7	125.1	765.6
MEAN	25	204.1222222	189.1222222	122.3555556	134.3666667	103	98.98888889	271.1888889
STANDARD DEVIATION		111.7368759	99.92115364	90.32654802	35.73566146	25.77241743	24.72500983	244.2386663
TOTAL MEAN		84.42314435						

Table 8.12. FPS for the GeForce GTX 1070 non-Ti in CPU simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size.

	Distance	CUMULUS (800 x 600)	CUMULUS (1200 x 600)	CUMULUS (1400 x 700)	METABALLS (1200 x 600)
	0	593.5	530	410.7	376.9
	10	219.7	238.6	90.7	130.6
	15	126.9	121.8	60.1	94.9
	20	101.2	97.5	53.5	104.5
	30	127.5	124.7	73	112.8
	35	154.9	146	90.2	151.2
	40	182.9	157.9	99.7	209.71
	50	238.8	196.4	120.6	868.2
	>100	338.2	294.6	191.5	1335.8
MEAN		231.5111111	211.9444444	132.2222222	376.0677778
STANDARD DEVIATION		153.942436	134.6514214	112.1675842	436.4472925
TOTAL MEAN		122.5074438			

Table 8.13. FPS benchmarks for the GeForce GTX 1070 non-Ti in CUDA simulation mode for a $Light = 10^3, Fluid = 100 \times 20 \times 40$ problem size.

	Distance	CUMULUS (800 x 600)	CUMULUS (1200 x 600)	CUMULUS (1400 x 700)	METABALLS (1200 x 600)
	0	128.5	124.4	134.8	279
	10	80.7	71.9	67.8	136.6
	15	68.4	67.4	60.5	79.7
	20	65.2	55.8	44.2	88
	30	65.1	65.9	53.9	116.4
	35	74.4	70.9	60.3	146
	40	78.5	74.9	65.7	156.9
	50	86	84.5	72.9	250.9
	>100	107.6	99.2	91.4	396.5
MEAN	25	83.82222222	79.43333333	72.38888889	183.3333333
STANDARD DEVIATION		21.29623076	20.841665	26.80151322	104.7271932
TOTAL MEAN		58.36273346			

Table 8.14. FPS benchmarks for the GeForce GTX 1070 non-Ti in CPU simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size.

	Distance	CUMULUS (800 x 600)	CUMULUS (1200 x 600)	CUMULUS (1400 x 700)	METABALLS (1200 x 600)
	0	546.2	401.5	476.3	495.6
	10	153.1	117.8	98.5	167
	15	109	101.8	82.7	86.9
	20	97.7	76.3	53.2	97.9
	30	99.7	103.4	75	138.4
	35	129.1	122.8	84.6	183
	40	139.6	133.6	98.3	201
	50	183.5	164.3	114.5	409
	>100	318.7	232.2	183.6	1164
MEAN	25	197.4	161.5222222	140.7444444	326.9777778
STANDARD DEVIATION		147.4095061	100.6173789	130.9659986	343.4449984
TOTAL MEAN		107.1395128			

Table 8.15. FPS metrics for the GeForce GTX 1070 non-Ti in CUDA simulation mode for a $Light = 40^3, Fluid = 100 \times 40 \times 40$ problem size.

8.3 CUDA parallel algorithm analysis

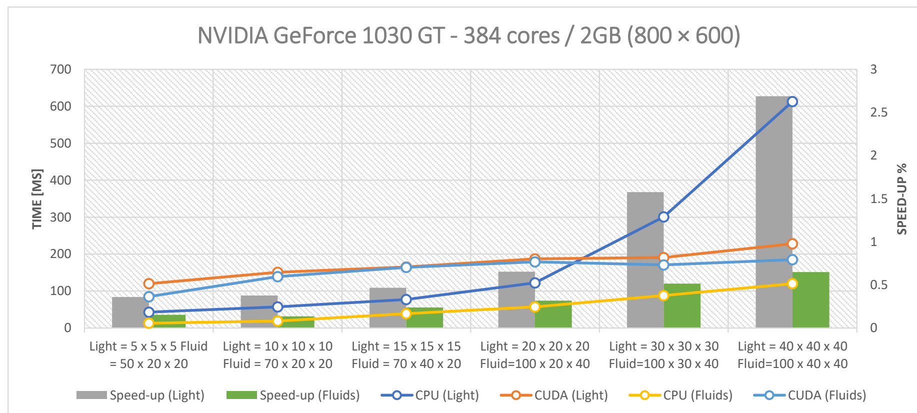


Figure 8.17. Speedup calculation for the nVidia 1030 GT. The CPU fluid version improves the CUDA performance.

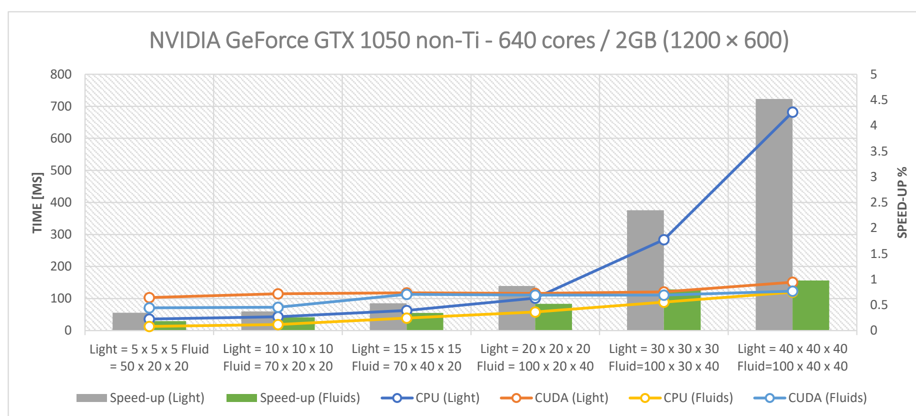


Figure 8.18. Speedup for the nVidia GTX 1050 non-Ti. The CPU version still performs better than the CUDA implementation.

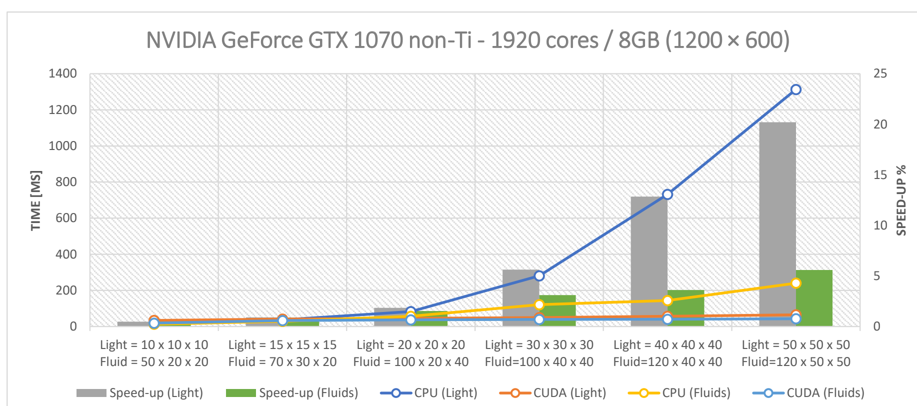


Figure 8.19. Speedup benchmarks for the nVidia GTX 1070 non-Ti. Both the fluid and the light CUDA versions improve the CPU performance considerably.

The metrics that Figures 8.17, 8.18 and 8.19 illustrate were measured with four dynamic cumulus clouds with different sizes for the precomputed light grid and fluid engine grid. As can be observed in the previous graphs, the speedup (see Section 2.4.2) increases as the problem size increases. This effect is demonstrated on the right side of the graphs when the CPU time for the precomputed light algorithm overcomes the CUDA version as measured in milliseconds. In this case, the CUDA implementation is much better in all studied graphics cards.

8.4 Quality of rendering

Figures 8.20 to 8.28 show a collection of different cloud types from high- to low-altitude clouds. They are the most reliable proof of the quality of the rendering that the implemented models researched in this thesis have achieved.

Figure 8.20. Cirrus Castellanus.

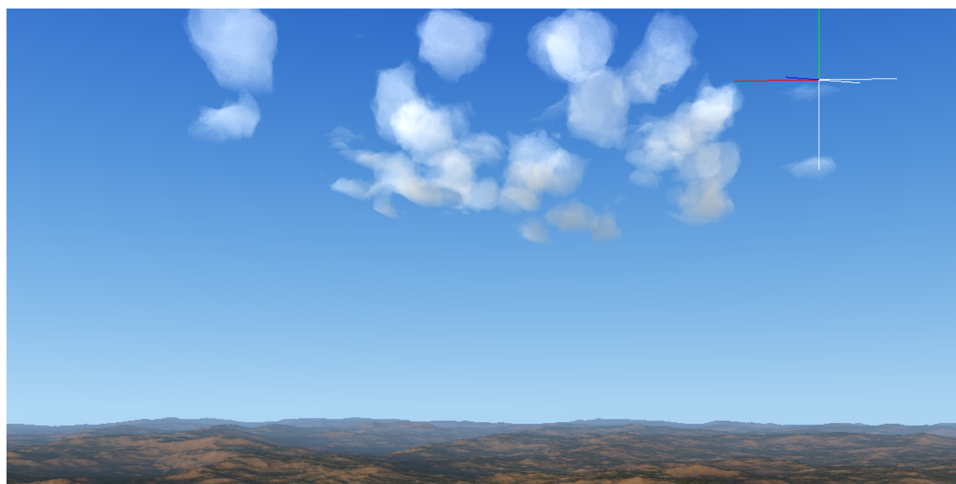


Figure 8.21. Cirrus Uncinus.

Figure 8.22. Altostratus Undulatus.

Figure 8.23. Altocumulus Lenticularis Duplicatus.

Figure 8.24. Altocumulus Castellanus.

Figure 8.25. Cumulus Humilis.

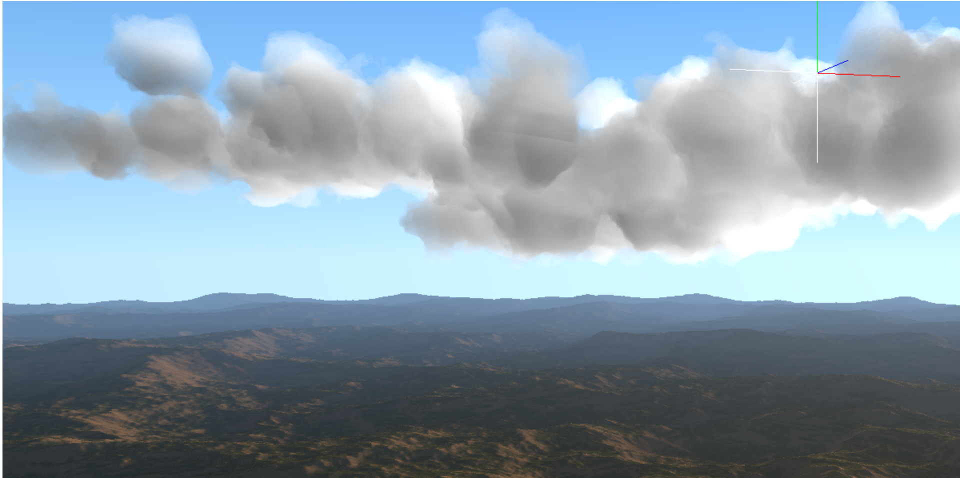


Figure 8.26. Stratocumulus.

Figure 8.27. Cumulonimbus Calvus.

Figure 8.28. Cumulonimbus Incus.

8.5 Summary

The exposed chapter is a complete analytical and empiric study of the presented algorithms and models. The first part of the chapter evaluates the complexity of the base algorithm for cloud rendering and morphing for its implementation in a GPU through GLSL shaders. This chapter also measures the times and FPS metrics in both serial and parallel code for static and dynamic rendering and points out the impact of the problem sizes in the serial (CPU) and CUDA overall performance. Finally, a wide collection of cloud types are illustrated to demonstrate the quality of the rendering.

CHAPTER 9

Discussions, conclusions and future work

THE final chapter of this thesis gives an overview of all the work carried out. Thus, Section 9.1 discusses the critical parts of this thesis in comparison to other authors' models, providing a clarification about the main achievements and the feature elements that are lacking. A review of the developed contents and accomplished features is summarized in Section 9.2 containing the conclusions. This work ends by proposing future possible improvements for the final user with the aim to complete the remaining features as explained in Section 9.3.

9.1 Discussions

In comparison with particle system and basic raytracing methods, the proposed algorithm results in greater realism than the one found in [KPK15; Hua+08; Har02; Bi+16; Hor+05; Mon+17], as shown in Figures 9.1, 9.2 and 9.3 and higher performance (Table 9.1). The obtained realism is comparable to that of off-line and photo-realistic models that use all physical characteristics, as shown in Figure 9.4, that are not suitable for use in real-time due to the long execution times required. A novel non-real-time approach is found in [Kal+17] which applies the radiance-predicting neural network model (RPNN) to emulate real cumuli. However, it takes ~ 12 hours to train the network using an nVidia Titan (Pascal) GPU and two Intel Xeon CPUs (24 cores/48 threads in total). The present research goes in the direction of balancing the realism of volumetric rendering and the performance of particle systems to exceeds the limitations of the non-real-time and photo-realistic methods cited in [Nar+06; Jar+08; Cer+05]. It is also a suitable framework for implementing lighting and shadowing algorithms, with lower GPU overhead as compared to the methods of other works [GMF09; Zho+08; Del+10]. The present research improves pre-computation times to the millisecond level as opposed to minutes [Yus14; Zho+08; ASW13], or hours [Kal+17].

With respect to the research by Bouthors [Bou+08], the thesis model shows an increase in FPS performance on the same graphics hardware. While not obtaining the hyperrealism of radiometry achieved in that model, the thesis research about cloud generation method using 3D meshes improves on the geometry accuracy and smoothness obtained by Wither et al. in the rapid sketch modelling of clouds [WBC08]. In contrast to the realistic method by Mukhina [MB15], where the cloud is projected on an hemispherical disc, the volumetric algorithm of this thesis allows for the navigation and traversal of gaseous mass to observe details. This thesis also improves on the overall realism in the model of Elek et al. [Ele+12], while maintaining the same performance as the conservative raymarching version of their implementation when it is executed at 1920×1080 pixels. Despite the efforts, the method of this thesis lacks the precision of the excellent work by Klehm et al. [KSE14] and Peters et al. [Pet+16] with respect to scattering and shadow maps.

None of the algorithms cited above covers the aspect of cloud motion and shape alteration. This thesis implements the effect of wind advection over a dynamic system of primitives efficiently as seen in Chapter 7.

Regarding the morphing algorithm, the proposed animation yields between 80 and 137 FPS with approximately 350 ellipsoids, which implies better performance and rendering quality than the work of [YW11]. This thesis also presents a new approach for cloud dynamics based on a GPGPU that achieves realistic cloud movement using parallel algorithms with a significant increment in speedup by applying CUDA technology in comparison to the outdated serial algorithms cited above. In addition, the guide points simplify the motion implementation and software engineering complexity in an affordable way, as opposed to other accurate but challenging-to-implement methods such as [Dob+00].

Figure 9.1. Generated cloud using a particle system with Harris [HL01], Huang et al. methods [Hua+08]. The contour of the cloud and the overall realism lack accuracy.

Figure 9.2. A cloud modelling method by Montenegro et al. [Mon+17] that combines procedural and implicit models.

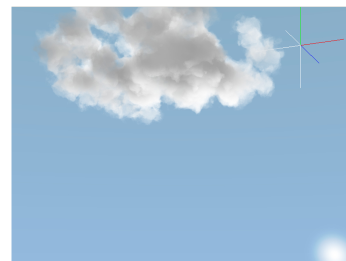


Figure 9.3. The proposed ontological volumetric cloud rendering method with lighting. The procedural noise improves cloud edges and fuzzy volume effects.

	Huang et al.	Montenegro et al.	Kang et al.	Yusov	Bi et al.	Present thesis
FPS	99.5	30	60	105 (GTX 680)	50	> 150 (GTX 1050 non-Ti)

Table 9.1. Average frequency of other particle and volumetric systems compared with the proposed method.

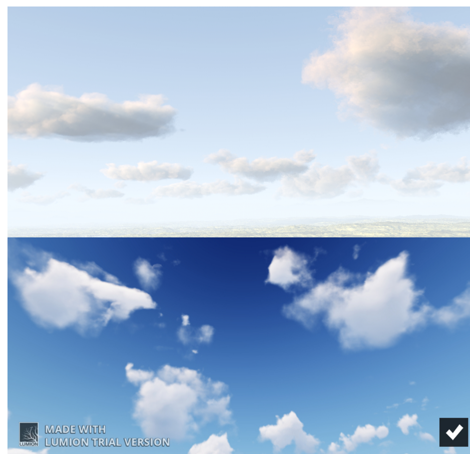


Figure 9.4. Photo-realistic clouds with all physical characteristics. The image above was generated with POV-Ray in 1 h, 18 min using the 100% of the CPU cores at 1024 × 768 resolution in pixels. The image below was generated with Lumion 7.5 taking around 1 s for the skydome texture. These are antithetical model examples that differ from the real-time system explained in this thesis.

9.2 Conclusions

This thesis presents a real-time cloud rendering method with a good balance between realism and performance. The algorithm uses flat uniform noise transformed into fBm with low memory usage ($64 \times 64 \times 64$ single-precision floats) to ensure efficient computational costs of raymarching. The constructive primitives are composed of a new particle called pseudosphere which equation has been developed hereby. In addition, the use of bounding boxes containing very few pseudospheroids allows for the application of a simple linear loop which discards clouds outside the camera view. This low number of pseudospheroids is achieved thanks to a custom developed Gaussian equation that covers the most typical cumuli used in video games and virtual reality software. The linear discrimination can perform even better if sphere location is made with space-partitioning algorithms, but it requires much coding time and complex CPU/GPU coordination.

The use of pre-calculated light stored in a voxel grid along with optimizations like the *no duplicate tracing* algorithm improves the speed of light computing. This algorithm is optimal when the location of the viewer changes faster than the location of the light source or the geometry of the clouds. A limitation of the lighting model is the calculation of multiple scattering only in the forward direction, which neglects 50% of the remaining scattered light.

The algorithm implements a function that allows the use of 3D meshes to define the shape of clouds that should resemble recognisable objects. A novel technique has been used, consisting of fitting ellipsoids to the vertices of each triangle and rotating them to smooth the look of the cloud by the use of Rodrigues' formula.

Regarding the motion and cloud dynamics, this thesis addresses the simplification of the Navier-Stokes fluid equations by making use of a reduced version of a fluid solver with CUDA parallel capabilities to adapt it to the previous static rendering research. Thus, the application of the novel technique of the guide points eases the implementation with very reliable results.

As an artistic and attractive effect, this thesis also presents a new morphing algorithm for 3D meshes evoking animated pareidolic phenomena.

In summary, taking into account the data presented in the results Chapter 8, it can be conveyed that the initial hypothesis is confirmed and that these algorithms are a good candidate for applications requiring a good balance between performance and realism, such as computer games, flight simulators, and virtual reality.

9.3 Future work

All the work performed in the course of this thesis aims to recreate clouds with all their beauty while keeping in mind the efficiency requirements. There is still room for future improvements as noted below:

- **A precipitation model:** It would be useful for a complete but efficient rain simulation model to be used in computer games and flight simulators. Such a model should recreate all precipitation artefacts, such as hail, snow, lightning and turbulences.
- **Shadow mapping:** The present implementation lacks projected shadows; therefore, the recreation of this effect would be advantageous.
- **Light beams:** This thesis also lacks light beams from either sunset or sunrise. Implementation of this effect would slow down the overall execution but is very feasible with the exposed model.
- **Full-featured cloud deformation:** The guide points technique should be applied to all the spheres of the cloud to achieve a more realistic turbulent deformation.

9.4 Summary

This chapter is the final point of the trip. The main pros and cons of the carried out research are discussed and analysed along with other works and models accomplished by authors worldwide. At last, a challenging message is given to the reader to highlight the applied efforts in the course of this thesis with the hope that others will continue this work in the future.

APPENDIX A

Appendix A

Listing A.1. Cumulus static and dynamic simplified shader.

```
////////////////////////////////////////////////////////////////////
// ONTOGENETIC MODEL FOR REAL-TIME VOLUMETRIC CLOUDS SIMULATION THESIS
// Software Engineering and Computer Systems Department
// National University for Distance Education (UNED)
// Carlos Jiménez de Parga, PhD student.
// License Creative Commons Attribution-NonCommercial-ShareAlike 3.0
// Unported License.
// Last revision 19/04/2019
//////////////////////////////////////////////////////////////////

#version 450 core
#pragma optimize(on)


// Declare uniforms

uniform sampler3D iNoise;
uniform sampler3D iVoxel[10];
uniform float iDepth;
uniform int iNumSph;
uniform int iNumClouds;
uniform bool isFlat[10];
uniform int iTurn;
```

```
24 uniform vec2 iResolution;
25 uniform mat4 iView;
26 uniform vec3 iPos;
27 uniform vec3 iVmin[10];
28 uniform vec3 iVmax[10];
29 uniform int iLowLimits[10];
30 uniform int iUpLimits[10];
31 uniform float iTime;
32 uniform vec3 iWindDirection;
33 uniform vec3 iSunDir;
34 uniform bool iSnow;
35
36 // Global variables
37
38 vec3 cloudColor;         // Color of the cloud
39 vec3 sunColor;           // Color of light source
40 vec4 candidates[100];    // List of candidates
41 float kPhase;            // Phase function constant
42 float T;                 // Light threshold
43 int n = 0;               // Number of candidates
44 vec3 rdNorm;             // Normalized ray-direction
45 vec3 skyColor;           // Color of the sky
46 float sunSize = 0.5;     // Sun radius
47 float tmin, tmax, tymin, tymax, tzmin, tzmax; // For Smits' algorithm
48
49 in vec2 fragCoord; // Fragment shader 2D coordinates
50 out vec4 FBColor;  // Returned frame buffer color
51
52 // Spheres array
53
54 layout(std140) uniform iCloudPosBlock
55 {
56     vec4 iCloudPos[150];
57 };
58
59 // Calculate fBm noise
60
61 float fbm(in vec3 q)
```

```
62 {
63     q = q + iWindDirection * iTime;
64
65     float f = 0.0;
66     float a = 0.5;
67
68     for (int i = 0; i < 5; i++) {
69         f += a * textureLod(iNoise, q / 128.0, -100.0).r;
70         q = q * 2.0 + 0.03;
71         a *= 0.5;
72     }
73
74     return f;
75 }
76
77 // Collision detection routine
78
79 void addCandidates(in vec3 rayOrg, in vec3 rayDir, int boundIdx)
80 {
81     // Limits of the cloud spheres
82
83     int sphStart = iLowLimits[boundIdx];
84     int sphEnd = iUpLimits[boundIdx];
85
86     // Iterate over spheres
87
88     for (int j = sphStart; j < sphEnd; j++)
89     {
90
91         vec3 cloudPos = iCloudPos[j].xyz; // Sphere (x,y,z) position
92         float radius = iCloudPos[j].w;   // Sphere radius
93
94         vec3 temp = rayOrg - cloudPos;
95         float a = dot(rayDir, rayDir);
96         float b = 2.0*dot(rayDir, temp);
97         float c = dot(temp, temp) - radius * radius;
98
99         float disc = b * b - 4.0*a*c;
```

```
100
101         if (disc > 0.0) // There is a collision
102         {
103
104             disc = sqrt(disc);
105             float t = ((-b - disc) / (2.0*a));
106             float limit = ((-b + disc) / (2.0*a));
107             candidates[n] = vec4(t, limit, j, boundIdx); // Add to candidates array
108             n++;
109
110         }
111     }
112 }
113
114 // Insertion-sort algorithm
115
116 void order()
117 {
118     int h;
119     vec4 aux;
120
121     for (int i = 1; i < n; i++)
122     {
123         aux = candidates[i];
124         h = i - 1;
125         while ((h >= 0) && (aux.x < candidates[h].x))
126         {
127             candidates[h + 1] = candidates[h];
128             h--;
129         }
130         candidates[h + 1] = aux;
131     }
132 }
133
134
135 // Simplified Henyey-Greenstein phase function
136
137 float phase(in float g)
```

```
138 {
139     return 0.0795 * ((1.0 - g * g) /
140      pow(1.0 + g * g - 2.0*g*dot(rdNorm, iSunDir), 1.5));
141 }
142
143 // Raytrace pseudo-sphere
144
145 vec4 trace(in vec3 rayOrg, in vec3 rayDir, in vec3 color)
146 {
147
148     float localDen, den;
149
150     float tIn, tOut;
151
152     for (int i = 0; i < n; i++)
153     {
154
155         int sphIdx = int(candidates[i].z);      // Sphere index
156         int boundIdx = int(candidates[i].w);    // Cloud index
157         vec3 cloudPos = iCloudPos[sphIdx].xyz; // Sphere 3D position
158         float radius = iCloudPos[sphIdx].w;    // Sphere radius
159         vec3 iVoxMin = iVmin[boundIdx];        // Bounding box 3D min values
160         vec3 iVoxMax = iVmax[boundIdx];        // Bounding box 3D max values
161         vec3 difVox = iVoxMax - iVoxMin;       // Voxel distance
162
163         tIn = candidates[i].x;                 // Lambda-in
164         tOut = candidates[i].y;                // Lambda-out
165
166
167         bool bFlat = isFlat[boundIdx];        // Is the cloud flat?
168
169         // LOD (Level-of-Detail)
170         float lambda = clamp(10.0*exp(-distance(rayOrg, cloudPos)*0.23), 0.1, 10.0);
171
172         while (tIn <= tOut) // Iterate over candidate sphere list
173         {
174             vec3  pos = rayOrg + tIn * rayDir; // Raymarch position
175
```

```
176             vec3 index = (pos - iVoxMin) / difVox; // Voxel index
177 
178             // Pre-compute light value
179             float precLight = texture(iVoxel[boundIdx], index, -100.0).r;
180 
181       // Condition for flat cloud
182             if (!bFlat || (pos.y > iVoxMin.y + precLight * 1.2))
183             {
184                 den = fbm(pos); // Density
185 
186                 // Create pseudo-sphere
187                 localDen = exp(-distance(pos, cloudPos) / (radius*(0.7 + 0.4*den)));
188 
189                 // Render pseudo-sphere with lighting
190                 if (den < localDen)
191                 {
192 
193                     float deltaT; // Calculate differential transmittance
194 
195                     deltaT = exp(-0.2 * den);
196 
197                     vec3 absorpLight = sunColor * precLight; // Absorption
198 
199                     // Scattering
200                     vec3 scatterLight = sunColor * phase(kPhase)*precLight;
201 
202                     // Total light
203                     vec3 totalLight = absorpLight * 0.9 + scatterLight;
204 
205                     // Resulting cloud color
206                     color += (1.0 - deltaT) * totalLight * T;
207 
208                     T *= deltaT; // Accumulate transmittance
209 
210                     if (T < 1e-6) // Exit condition
211                         return vec4(color, 1 - T);
212 
213                 }
```

```
214             }
215
216             tIn += lambda; // Step raymarching
217
218         }
219
220     }
221
222     return vec4(color, 1 - T); // Return color with transparency
223 }
224
225 // Render cloud
226
227 vec4 render(in vec3 rayOrg, in vec3 rayDir, in vec2 px)
228 {
229
230     rdNorm = normalize(rayDir);
231
232     // background sky
233     vec4 skycol;
234     vec4 res = vec4(0);
235
236     float sun = 1.0 + dot(iSunDir, rdNorm);
237
238     if (iTurn == 0) // Morning scene
239     {
240         float fexp = exp(-sun * 1000.0 / sunSize);
241         float fexp2 = exp(-sun * 1000.0 / 0.1);
242
243         skycol = vec4(0.2*sunColor*fexp + 0.6*sunColor*fexp2, 1.0);
244         // sun glare
245         skycol += vec4(getSkyColor(rdNorm.xyz) + 0.2*sunColor*exp(-sun), 0);
246
247     }
248
249     else if (iTurn == 1) // Sunset scene
250     {
251
```

```
252            vec3 stars;
253
254            if ((rdNorm.y > 0.1) && (dot(iSunDir, rdNorm) > -0.995))
255            {
256                float g = 0.2*fbmgal(rdNorm); // Generate stars
257                stars = 0.3*vec3(g*g*g, g*g*1.3, 8.5*g);
258                if (length(stars) < 2.4)
259                    stars = vec3(0);
260            }
261
262            // sun glare
263            skycol = vec4(getSunset(rdNorm, px) +
264             0.2*sunColor*exp(-sun) + stars, 1.0);
265
266            resM.xyz = resM.xyz / 5.0; // Mountain darkness
267
268        }
269        else // Night scene
270        {
271            vec3  stars = vec3(0);
272
273            if ((dot(iSunDir, rdNorm) > -0.9988))
274            {
275                float g = 0.2*fbmgal(rdNorm); // Generate stars
276                stars = 0.3*vec3(g*g*g, g*g*1.3, 1.5*g);
277            }
278
279
280 #ifdef FULLMOON
281
282            // Render fullmoon
283            float fexp = exp(-sun * 1000.0 / sunSize);
284            float fexp2 = exp(-sun * 1000.0 / 0.2);
285
286
287            skycol = vec4(0.2*sunColor*fexp + sunColor * fexp2, 1.0);
288            // moon glare
289            skycol += vec4(0.2*sunColor*exp(-sun) + stars, 0);
```

```
290
291 #else
292
293         // Render quarter moon
294         float time = 812.81;
295
296         //Diffuse
297         float r = 0.05;
298         vec3 vp = vec3(sin(time*0.2), cos(time*0.2), sin(time*0.2));
299         vec3 vl = normalize(vp);
300         float diffuse = diffuseSphere(rdNorm, -iSunDir, r, vl);
301
302         skycol = vec4(vec3(diffuse) + stars, 1);
303 #endif
304
305         resM.xyz = resM.xyz / 8.0; // Mountain darkness
306     }
307
308     // Smits' algorithm
309
310     for (int i = 0; i < iNumClouds; i++)
311     {
312
313         float vminX = iVmin[i].x;
314         float vmaxX = iVmax[i].x;
315         float vminY = iVmin[i].y;
316         float vmaxY = iVmax[i].y;
317         float vminZ = iVmin[i].z;
318         float vmaxZ = iVmax[i].z;
319
320         bool flag = true;
321
322         if (rayDir.x >= 0)
323         {
324             tmin = (vminX - rayOrg.x) / rayDir.x;
325             tmax = (vmaxX - rayOrg.x) / rayDir.x;
326         }
327         else
```

```
328         {
329             tmin = (vmaxX - rayOrg.x) / rayDir.x;
330             tmax = (vminX - rayOrg.x) / rayDir.x;
331         }
332         if (rayDir.y >= 0)
333         {
334             tymin = (vminY - rayOrg.y) / rayDir.y;
335             tymax = (vmaxY - rayOrg.y) / rayDir.y;
336         }
337         else
338         {
339             tymin = (vmaxY - rayOrg.y) / rayDir.y;
340             tymax = (vminY - rayOrg.y) / rayDir.y;
341         }
342
343         if ((tmin > tymax) || (tymin > tmax))
344             flag = false;
345
346         if (tymin > tmin)
347             tmin = tymin;
348
349         if (tymax < tmax)
350             tmax = tymax;
351
352         if (rayDir.z >= 0)
353         {
354             tzmin = (vminZ - rayOrg.z) / rayDir.z;
355             tzmax = (vmaxZ - rayOrg.z) / rayDir.z;
356         }
357         else
358         {
359             tzmin = (vmaxZ - rayOrg.z) / rayDir.z;
360             tzmax = (vminZ - rayOrg.z) / rayDir.z;
361         }
362
363         if ((tmin > tzmax) || (tzmin > tmax))
364             flag = false;
365
```

```
366         // In bounding-box
367         if (flag)
368             addCandidates(rayOrg, rayDir, i);
369
370     }
371
372     T = 1.0; // Initialize T
373
374     if (n > 0 && resM.w < 0.0) // No mountain collision
375     {
376         order();
377         vec3 color = vec3(0.0);
378         res = trace(rayOrg, rayDir, color);
379
380         return res + skycol * T;
381     }
382     else if (n > 0 && resM.w > 0.0) // Mountain collision
383     {
384         order();
385         vec3 color = vec3(0.0);
386         res = trace(rayOrg, rayDir, color);
387
388         return resM * T + res;
389     }
390     else if (n == 0 && resM.w < 0.0)
391         return skycol;
392     else if (n == 0 && resM.w > 0.0)
393         return resM;
394 }
395
396
397
398
399
400
401
402
403
```

```
404
405
406 void main()
407 {
408
409     // Change coordinate system
410     vec2 p = (-iResolution.xy + 2.0*fragCoord.xy) / iResolution.y;
411
412     vec2 position = fragCoord.xy / iResolution.x;
413
414     switch (iTurn)
415     {
416     case 0:
417         // MORNING
418         sunColor = vec3(1.0, 1.0, 1.0);
419         kPhase = -0.4;
420         break;
421     case 1:
422         // SUNSET
423         sunColor = vec3(1.0, 0.8, 0.2);
424         kPhase = -0.4;
425         break;
426     case 2:
427         // NIGHT
428         sunColor = vec3(0.9, 0.9, 0.9);
429         kPhase = -0.38;
430     }
431
432     // Camera ray
433     vec4 rayDir = iView * normalize(vec4(p, 1.5, 1));
434
435     // Frame-buffer return color
436     FBColor = render(iPos, rayDir.xyz, position);
437
438 }
```

APPENDIX B

Appendix B

Listing B.1. Mesh/Morphing static and dynamic simplified shader.

```
////////////////////////////////////////////////////////////////////
// ONTOGENETIC MODEL FOR REAL-TIME VOLUMETRIC CLOUDS SIMULATION THESIS
// Software Engineering and Computer Systems Department
// National University for Distance Education (UNED)
// Carlos Jiménez de Parga, PhD student.
// License Creative Commons Attribution-NonCommercial-ShareAlike 3.0
// Unported License.
// Last revision 19/04/2019
//////////////////////////////////////////////////////////////////

#version 450 core
#pragma optimize(on)

// Declare uniforms

uniform sampler3D iNoise;
uniform sampler3D iVoxel[10];
uniform float iDepth;
uniform int iNumSph;
uniform int iNumClouds;
uniform int iTurn;
uniform vec2 iResolution;
uniform mat4 iView;
```

```
24 uniform vec3 iPos;
25 uniform vec3 iVmin[10];
26 uniform vec3 iVmax[10];
27 uniform float iNumVoxel;
28 uniform int iLowLimits[10];
29 uniform int iUpLimits[10];
30 uniform int iDebug;
31 uniform float iTime;
32 uniform vec3 iWindDirection;
33 uniform float iAlpha;
34 uniform bool iEvolute;
35 uniform bool iSnow;
36 uniform vec3 iSunDir;
37
38 // Global variables
39
40 vec3 cloudColor;          // Color of the cloud
41 vec3 sunColor;            // Color of light source
42 vec4 candidates[100];     // List of candidates
43 float kPhase;             // Phase function constant
44 float T;                  // Light threshold
45 int n = 0;                // Number of candidates
46 vec3 rdNorm;              // Normalized ray-direction
47 vec3 skyColor;            // Color of the sky
48 float sunSize = 0.5;      // Sun radius
49 float tmin, tmax, tymin, tymax, tzmin, tzmax; // For Smits' algorithm
50
51 in vec2 fragCoord; // Fragment shader 2D coordinates
52 out vec4 FBColor;  // Returned frame buffer color
53
54 int vSrc, vDst;   // Source and destination meshes
55
56 // Ellipsoid array
57
58 uniform iCloudPosBlock
59 {
60     mat4 iCloudPos[500];
61 };
```

```
62
63 // Rodrigues' rotation array
64
65 uniform iCloudPosBlockR
66 {
67     mat4 iCloudPosR[500];
68 };
69
70 // Calculate fBm noise
71
72 float fbm(in vec3 q)
73 {
74     q = q - vec3(0.0, 0.1, 1.0)*iTime;
75
76     float f = 0.0;
77     float a = 0.5;
78
79     for (int i = 0; i < 5; i++) {
80         f += a * texture(iNoise, q / 256.0, -100.0).r;
81         q = q * 2.0 + 0.03;
82         a *= 0.5;
83     }
84
85     return f;
86 }
87
88 // Collision detection routine
89
90 void addCandidates(in vec3 rayOrg, in vec3 rayDir, int boundIdx)
91 {
92
93     // Limits of the cloud ellipsoids
94
95     int sphStart = iLowLimits[boundIdx];
96     int sphEnd = iUpLimits[boundIdx];
97
98     // Iterate over ellipsoids
99
```

```
100     for (int j = sphStart; j < sphEnd; j++)
101     {
102
103         float cx = iCloudPos[j][0].x; // Ellipsoid position
104         float cy = iCloudPos[j][0].y;
105         float cz = iCloudPos[j][0].z;
106
107         float a = iCloudPos[j][1].x; // Ellipsoid radius
108         float b = iCloudPos[j][1].y;
109         float c = iCloudPos[j][1].z;
110
111         float x0 = rayOrg.x;
112         float y0 = rayOrg.y;
113         float z0 = rayOrg.z;
114
115         float vx = rayDir.x;
116         float vy = rayDir.y;
117         float vz = rayDir.z;
118
119         float k11, k12, k13, k21, k22, k23, k31, k32, k33;
120
121         mat4 R = iCloudPosR[j];
122
123         k11 = R[0].x;
124         k12 = R[0].y;
125         k13 = R[0].z;
126
127         k21 = R[1].x;
128         k22 = R[1].y;
129         k23 = R[1].z;
130
131         k31 = R[2].x;
132         k32 = R[2].y;
133         k33 = R[2].z;
134
135         // Discriminant
136         float disc = -(1.0 / (a*a)*pow(k11*vx + k12 * vy + k13 * vz, 2.0) +
137          1.0 / (b*b)*pow(k21*vx + k22 * vy + k23 * vz, 2.0) +
```

```
138            1.0 / (c*c)*pow(k31*vx + k32 * vy + k33 * vz, 2.0))*
139            (1.0 / (a*a)*pow(k11*(cx - x0) +  k12 * (cy - y0) +
140            k13 * (cz - z0), 2.0) + 1.0 / (b*b)*pow(k21*(cx - x0) +
141            k22 * (cy - y0) + k23 * (cz - z0), 2.0) + 1.0 /
142            (c*c)*pow(k31*(cx - x0) + k32 * (cy - y0) +
143             k33 * (cz - z0), 2.0) - 1.0) + pow(1.0 / (a*a)*(k11*vx*2.0 +
144            k12 * vy*2.0 + k13 * vz*2.0)*(k11*(cx - x0) + k12 *
145            (cy - y0) + k13 * (cz - z0)) + 1.0 / (b*b)*(k21*vx*2.0 +
146            k22 * vy*2.0 + k23 * vz*2.0)*(k21*(cx - x0) + k22 *
147            (cy - y0) + k23 * (cz - z0)) + 1.0 / (c*c)*
148            (k31*vx*2.0 + k32 * vy*2.0 + k33 * vz*2.0)*
149            (k31*(cx - x0) + k32 * (cy - y0) + k33 * (cz - z0)), 2.0)*
150            0.25;
151
152
153           if (disc > 0) // There is a collision with the ellipsoid
154           {
155               float sqr = sqrt(disc);
156
157               float num = (a*a)*(b*b)*cx*(k31*k31)*vx + (a*a)*(b*b)*
158               cy*(k32*k32)*vy +    (a*a)*(b*b)*cz*(k33*k33)*vz +
159               (a*a)*(c*c)*cx*(k21*k21)*vx + (a*a)*(c*c)*cy*(k22*k22)*vy +
160               (a*a)*(c*c)*cz*(k23*k23)*vz + (b*b)*(c*c)*cx*(k11*k11)*vx +
161               (b*b)*(c*c)*cy*(k12*k12)*vy + (b*b)*(c*c)*cz*(k13*k13)*vz -
162               (a*a)*(b*b)*(k31*k31)*vx*x0 - (a*a)*(c*c)*(k21*k21)*vx*x0 -
163               (a*a)*(b*b)*(k32*k32)*vy*y0 - (b*b)*(c*c)*(k11*k11)*vx*x0 -
164               (a*a)*(c*c)*(k22*k22)*vy*y0 - (a*a)*(b*b)*(k33*k33)*vz*z0 -
165               (b*b)*(c*c)*(k12*k12)*vy*y0 - (a*a)*(c*c)*(k23*k23)*vz*z0 -
166               (b*b)*(c*c)*(k13*k13)*vz*z0 + (a*a)*(b*b)*cx*k31*k32*vy +
167               (a*a)*(b*b)*cy*k31*k32*vx + (a*a)*(b*b)*cx*k31*k33*vz +
168               (a*a)*(b*b)*cz*k31*k33*vx + (a*a)*(b*b)*cy*k32*k33*vz +
169               (a*a)*(b*b)*cz*k32*k33*vy + (a*a)*(c*c)*cx*k21*k22*vy +
170               (a*a)*(c*c)*cy*k21*k22*vx + (a*a)*(c*c)*cx*k21*k23*vz +
171               (a*a)*(c*c)*cz*k21*k23*vx + (a*a)*(c*c)*cy*k22*k23*vz +
172               (a*a)*(c*c)*cz*k22*k23*vy + (b*b)*(c*c)*cx*k11*k12*vy +
173               (b*b)*(c*c)*cy*k11*k12*vx + (b*b)*(c*c)*cx*k11*k13*vz +
174               (b*b)*(c*c)*cz*k11*k13*vx + (b*b)*(c*c)*cy*k12*k13*vz +
175               (b*b)*(c*c)*cz*k12*k13*vy - (a*a)*(b*b)*k31*k32*vy*x0 -
```

```
176             (a*a)*(b*b)*k31*k33*vz*x0 - (a*a)*(b*b)*k31*k32*vx*y0 -
177             (a*a)*(c*c)*k21*k22*vy*x0 - (a*a)*(c*c)*k21*k23*vz*x0 -
178             (a*a)*(b*b)*k32*k33*vz*y0 - (a*a)*(c*c)*k21*k22*vx*y0 -
179             (a*a)*(b*b)*k31*k33*vx*z0 - (b*b)*(c*c)*k11*k12*vy*x0 -
180             (a*a)*(b*b)*k32*k33*vy*z0 - (b*b)*(c*c)*k11*k13*vz*x0 -
181             (a*a)*(c*c)*k22*k23*vz*y0 - (b*b)*(c*c)*k11*k12*vx*y0 -
182             (a*a)*(c*c)*k21*k23*vx*z0 - (a*a)*(c*c)*k22*k23*vy*z0 -
183             (b*b)*(c*c)*k12*k13*vz*y0 - (b*b)*(c*c)*k11*k13*vx*z0 -
184             (b*b)*(c*c)*k12*k13*vy*z0;
185
186             float denom = (a*a)*(b*b)*(k31*k31)*(vx*vx) +
187             (a*a)*(b*b)*(k32*k32)*(vy*vy) + (a*a)*(b*b)*(k33*k33)*(vz*vz) +
188             (a*a)*(c*c)*(k21*k21)*(vx*vx) + (a*a)*(c*c)*(k22*k22)*(vy*vy) +
189             (a*a)*(c*c)*(k23*k23)*(vz*vz) + (b*b)*(c*c)*(k11*k11)*(vx*vx) +
190             (b*b)*(c*c)*(k12*k12)*(vy*vy) + (b*b)*(c*c)*(k13*k13)*(vz*vz) +
191             (a*a)*(b*b)*k31*k32*vx*vy*2.0 + (a*a)*(b*b)*k31*k33*vx*vz*2.0 +
192             (a*a)*(b*b)*k32*k33*vy*vz*2.0 + (a*a)*(c*c)*k21*k22*vx*vy*2.0 +
193             (a*a)*(c*c)*k21*k23*vx*vz*2.0 + (a*a)*(c*c)*k22*k23*vy*vz*2.0 +
194             (b*b)*(c*c)*k11*k12*vx*vy*2.0 + (b*b)*(c*c)*k11*k13*vx*vz*2.0 +
195             (b*b)*(c*c)*k12*k13*vy*vz*2.0;
196
197             float prefix = a * a*b*b*c*c*sqr;
198             float t1 = (prefix + num) / denom;
199             float t2 = -(prefix - num) / denom;
200
201             float t, limit;
202
203             if (t1 < t2) // Order collision points
204             {
205                 t = t1;
206                 limit = t2;
207             }
208             else
209             {
210                 t = t2;
211                 limit = t1;
212             }
213
```

```
214                 // Add to candidates list
215                 candidates[n] = vec4(t, limit, j, boundIdx);
216                 n++;
217 
218             }
219         }
220 
221 }
222 
223 // Insertion-sort algorithm
224 
225 void order()
226 {
227     int h;
228     vec4 aux;
229 
230     for (int i = 1; i < n; i++)
231     {
232         aux = candidates[i];
233         h = i - 1;
234         while ((h >= 0) && (aux.x < candidates[h].x))
235         {
236             candidates[h + 1] = candidates[h];
237             h--;
238         }
239         candidates[h + 1] = aux;
240     }
241 }
242 
243 // Simplified Henyey-Greenstein phase function
244 
245 float phase(in float g)
246 {
247     return 0.0795 * ((1.0 - g * g) /
248      pow(1.0 + g * g - 2.0*g*dot(rdNorm, iSunDir), 1.5));
249 }
250 
251 // Raytrace pseudo-ellipsoid
```

```
252
253 vec4 trace(in vec3 rayOrg, in vec3 rayDir, in vec3 color)
254 {
255
256     float localDen, den;
257
258     float tIn, tOut;
259
260     float t;
261
262     t = candidates[0].x;
263
264     for (int i = 0; i < n; i++)
265     {
266
267         int sphIdx = int(candidates[i].z);        // Ellipsoid index
268         int boundIdx = int(candidates[i].w);      // Bounding box index
269         vec3 cloudPos = iCloudPos[sphIdx][0].xyz; // Ellipsoid 3D position
270         float radius = iCloudPos[sphIdx][2].x;
271
272         // Bounding-boxes linear interpolation
273         vec3 iVoxMin = mix(iVmin[vSrc], iVmin[vDst], iAlpha);
274         vec3 iVoxMax = mix(iVmax[vSrc], iVmax[vDst], iAlpha);
275
276
277         tIn = t;
278         tOut = candidates[i].y;
279
280         // Iterate pseudo-ellipsoid
281         while (tIn <= tOut)
282         {
283             vec3  pos = rayOrg + tIn * rayDir; // Raymarching position
284
285             den = fbm(pos); // Density
286
287             localDen = exp(-distance(pos, cloudPos) / radius);
288
289             if (den < localDen) // Trace pseudo-ellipsoid
```

```
290             {
291
292                 // Calculate differential transmittance
293                 float deltaT = exp(-0.3*den);
294                 vec3 index = (pos - iVoxMin) / (iVoxMax - iVoxMin); //Voxel index
295
296                 // Interpolate bounding-boxes pre-computed light
297                 float precLight = mix(texture(iVoxel[vSrc], index, -100.0).r,
298                  texture(iVoxel[vDst], index, -100.0).r, iAlpha);
299
300
301                 vec3 absorpLight = sunColor * precLight; // Absorption
302                 // Scattering
303                 vec3 scatterLight = sunColor * phase(kPhase)*precLight;
304                 // Total light
305                 vec3 totalLight = absorpLight * 0.9 + scatterLight;
306                 // Resulting cloud color
307                 color += (1.0 - deltaT) * totalLight * T;
308
309                 T *= deltaT; // Accumulate transmittance
310
311                 if (T < 1e-6) // Exit condition
312                      return vec4(color, 1 - T);
313          }
314
315        tIn += 0.1; // Step raymarching
316     }
317
318 }
319
320 return vec4(color, 1 - T); // Return color with transparency
321 }
322
323 // Render cloud mesh
324
325 vec4 render(in vec3 rayOrg, in vec3 rayDir, in vec2 position, in vec4 resMount)
326 {
327     rdNorm = normalize(rayDir);
```

```
328
329     // Background sky
330     vec4 skycol;
331     vec4 res = vec4(0);
332
333     float sun = 1.0 + dot(iSunDir, rdNorm);
334
335     if (iTurn == 0) // Morning scene
336     {
337         float fexp = exp(-sun * 1000.0 / sunSize);
338         float fexp2 = exp(-sun * 1000.0 / 0.1);
339
340         skycol = vec4(0.2*sunColor*fexp + 0.6*sunColor*fexp2, 1.0);
341         // sun glare
342         skycol += vec4(getSkyColor(rdNorm.xyz) + 0.2*sunColor*exp(-sun), 0);
343
344     }
345
346     else if (iTurn == 1) // Sunset scene
347     {
348
349         vec3 stars;
350
351         if ((rdNorm.y > 0.1) && (dot(iSunDir, rdNorm) > -0.995))
352         {
353             // Create stars
354             float g = 0.2*fbmgal(rdNorm);
355             stars = 0.3*vec3(g*g*g, g*g*1.3, 8.5*g);
356             if (length(stars) < 2.4)
357                 stars = vec3(0);
358         }
359
360         // sun glare
361         skycol = vec4(getSunset(rdNorm, position) +
362         0.2*sunColor*exp(-sun) + stars, 1.0);
363
364         resM.xyz = resM.xyz / 5.0; // Mountain darkness
365
```

```
366     }
367     else // Night scene
368     {
369         vec3  stars = vec3(0);
370 
371 
372         if ((dot(iSunDir, rdNorm) > -0.9988))
373         {
374             // Create stars
375             float g = 0.2*fbmgal(rdNorm);
376             stars = 0.3*vec3(g*g*g, g*g*1.3, 1.5*g);
377         }
378 
379 #ifdef FULLMOON
380 
381         float fexp = exp(-sun * 1000.0 / sunSize);
382         float fexp2 = exp(-sun * 1000.0 / 0.2);
383 
384 
385         skycol = vec4(0.2*sunColor*fexp + sunColor * fexp2, 1.0);
386         // moon glare
387         skycol += vec4(0.2*sunColor*exp(-sun) + stars, 0);
388 
389 #else
390 
391         float time = 812.81;
392 
393 
394         //Diffuse
395         float r = 0.05;
396         vec3 vp = vec3(sin(time*0.2), cos(time*0.2), sin(time*0.2));
397         vec3 vl = normalize(vp);
398         float diffuse = diffuseSphere(rdNorm, -iSunDir, r, vl);
399 
400         skycol = vec4(vec3(diffuse) + stars, 1);
401 #endif
402 
403         resM.xyz = resM.xyz / 8.0; // Mountain darkness
```

```
404     }
405
406     // Smits' algorithm
407
408
409     float vminX = iVmin[2].x;
410     float vmaxX = iVmax[2].x;
411     float vminY = iVmin[2].y;
412     float vmaxY = iVmax[2].y;
413     float vminZ = iVmin[2].z;
414     float vmaxZ = iVmax[2].z;
415
416     bool flag = true;
417
418     if (rayDir.x >= 0)
419     {
420         tmin = (vminX - rayOrg.x) / rayDir.x;
421         tmax = (vmaxX - rayOrg.x) / rayDir.x;
422     }
423     else
424     {
425         tmin = (vmaxX - rayOrg.x) / rayDir.x;
426         tmax = (vminX - rayOrg.x) / rayDir.x;
427     }
428     if (rayDir.y >= 0)
429     {
430         tymin = (vminY - rayOrg.y) / rayDir.y;
431         tymax = (vmaxY - rayOrg.y) / rayDir.y;
432     }
433     else
434     {
435         tymin = (vmaxY - rayOrg.y) / rayDir.y;
436         tymax = (vminY - rayOrg.y) / rayDir.y;
437     }
438
439     if ((tmin > tymax) || (tymin > tmax))
440         flag = false;
441
```

```
442     if (tymin > tmin)
443         tmin = tymin;
444
445     if (tymax < tmax)
446         tmax = tymax;
447
448     if (rayDir.z >= 0)
449     {
450         tzmin = (vminZ - rayOrg.z) / rayDir.z;
451         tzmax = (vmaxZ - rayOrg.z) / rayDir.z;
452     }
453     else
454     {
455         tzmin = (vmaxZ - rayOrg.z) / rayDir.z;
456         tzmax = (vminZ - rayOrg.z) / rayDir.z;
457     }
458
459     if ((tmin > tzmax) || (tzmin > tmax))
460         flag = false;
461
462     // clouds
463     if (flag)
464         addCandidates(rayOrg, rayDir, 0);
465
466     if (n > 0 && resMount.w < 0.0) // No mountain collision
467     {
468         T = 1.0;
469         order();
470         vec3 color = vec3(0.0);
471         res = trace(rayOrg, rayDir, color);
472         return vec4((T == 1.0) ? skycol : res + skycol * T);
473     }
474     else if (n > 0 && resM.w > 0.0) // Mountain collision
475     {
476         T = 1.0;
477         order();
478         vec3 color = vec3(0.0);
479         res = trace(rayOrg, rayDir, color);
```

```
480             return vec4((T == 1.0) ? resMount : resMount * T + res);
481         }
482         else if (n == 0 && resMount.w < 0.0)
483             return skycol;
484         else if (n == 0 && resMount.w > 0.0)
485             return resMount;
486 
487 }
488 
489 void main()
490 {
491 
492     // Change coordinate system
493     vec2 p = (-iResolution.xy + 2.0*fragCoord.xy) / iResolution.y;
494     // ray
495 
496     vec2 position = fragCoord.xy / iResolution.x;
497 
498     switch (iTurn)
499     {
500     case 0:
501         // MORNING
502 
503         sunColor = vec3(1.0, 1.0, 1.0);
504         kPhase = -0.3;
505 
506         break;
507 
508     case 1:
509         // SUNSET
510         sunColor = vec3(1.0, 0.8, 0.2);
511         kPhase = -0.4;
512         break;
513     case 2:
514         // NIGHT
515         sunColor = vec3(0.8, 0.8, 0.8);
516         kPhase = -0.3;
517 
```

```
518             break;
519         }
520 
521         // Select either mesh evolution or involution
522         if (iEvolute)
523         {
524             vSrc = 0;
525             vDst = 1;
526         }
527         else
528         {
529             vSrc = 1;
530             vDst = 0;
531         }
532 
533         vec4 rayDir = iView * normalize(vec4(p, 1.5, 1));
534 
535         // Frame-buffer return color
536         FBColor = render(iPos, rayDir.xyz, position);
537 
538     }
```

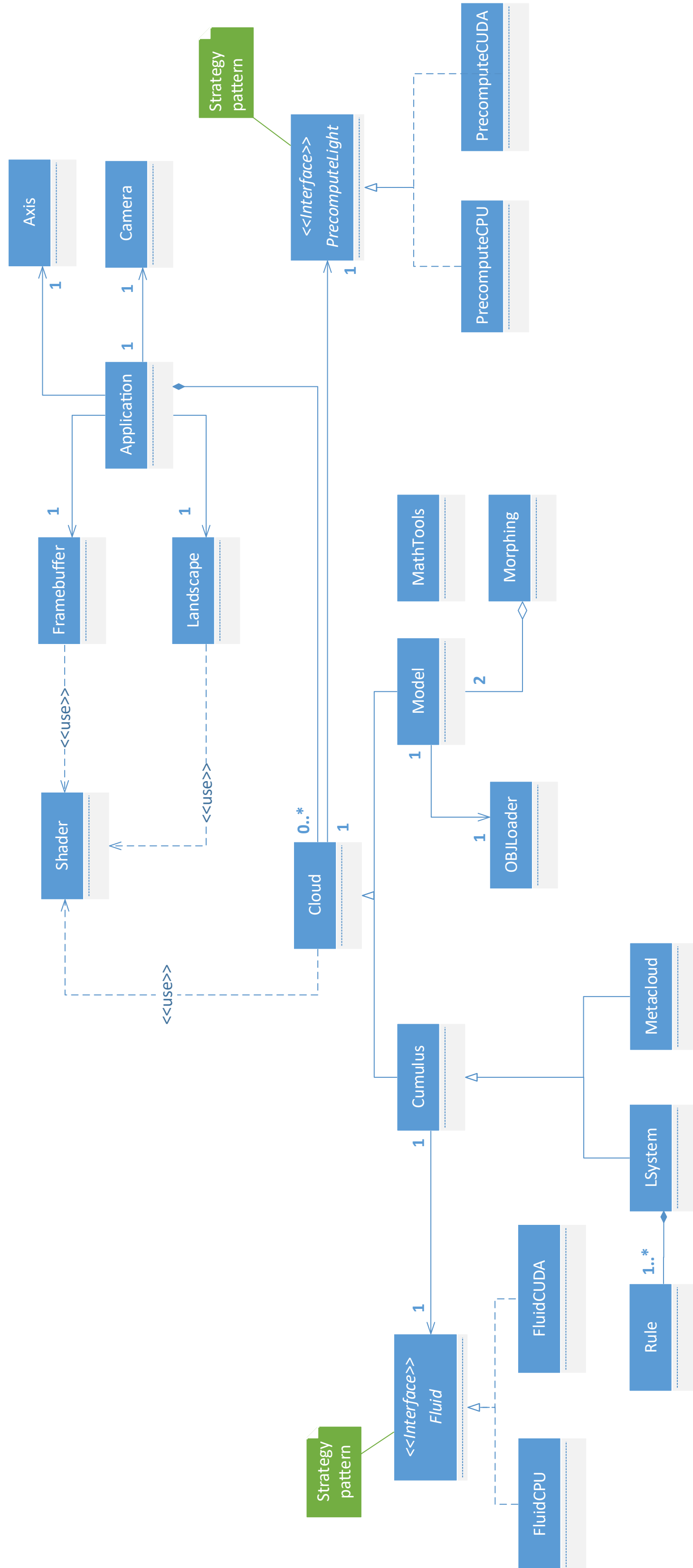


Figure C.1. Nimbus application framework class diagram.

Bibliography and references

[AG00] A. Apodaca and L. Gritz. *Advanced Renderman. Creating CGI for Motion Pictures.* Morgan Kaufmann, 2000 (cit. on p. 63).

[Ald97] J. Aldrich. “R.A. Fisher and the making of maximum likelihood 1912-1922”. In: *Statist. Sci.* 12.3 (Sept. 1997), pp. 162–176. DOI: 10.1214/ss/1030037906. URL: https://doi.org/10.1214/ss/1030037906 (cit. on p. 84).

[Ama09] G.N.P. Amador. “Real-time 3D rendering of water using CUDA”. MA thesis. 2009 (cit. on pp. 97, 99, 103).

[And10] D. Andrews. *An Introduction to Atmospheric Physics. Second Edition*. Cambridge University Press, 2010 (cit. on pp. 36, 37, 46).

[App68] A. Appel. “Some techniques for shading machine renderings of solids”. In: *Proc. AFIPS ’68 (Spring)*. Vol. 32. 1968, pp. 37–45. DOI: 10.1145/1468075.1468082 (cit. on pp. 2, 12).

[ASW13] M. Ament, F. Sadlo, and D. Weiskopf. “Ambient Volume Scattering”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.12 (Oct. 2013), pp. 2936–2945. DOI: 10.1109/TVCG.2013.129 (cit. on p. 135).

[BB96] G. Brassard and P. Bratley. *Fundamentals of algorithmics*. Vol. 524. Prentice Hall Englewood Cliffs, 1996 (cit. on pp. 74, 115).

[Bek19] R. Bekerman. *Ronenbekerman. Architectural Visualization Blog*. 2019. URL: https://www.ronenbekerman.com/lumion-realtime-architectural-visualization-tool-episode-1/ (cit. on p. 50).

[Bi+16] S. Bi et al. “3-Dimensional Modeling and Simulation of the Cloud Based on Cellular Automata and Particle”. In: *ISPRS international journal of geo-information* 5.6 (June 2016), p. 86. DOI: 10.3390/ijgi5060086 (cit. on p. 135).

[Bli82] J.F. Blinn. "A Generalization of Algebraic Surface Drawing". In: *ACM Transactions on Graphics* 1.3 (July 1982), pp. 235–256. DOI: 10.1145/357306.357310 (cit. on p. 90).

[BN92] T. Beier and S. Neely. "Feature-based Image Metamorphosis". In: *SIGGRAPH Comput. Graph.* 26.2 (July 1992), pp. 35–42. ISSN: 0097-8930. DOI: 10.1145/142920.134003. URL: http://doi.acm.org/10.1145/142920.134003 (cit. on p. 108).

[Bou+08] A. Bouthors et al. "Interactive multiple anisotropic scattering in clouds". In: *International Journal of Multimedia and Ubiquitous Engineering* (Feb. 2008), pp. 173–182. DOI: 10.1145/1342250.1342277 (cit. on pp. 117, 118, 136).

[Bou08] A. Bouthors. "Realistic rendering of clouds in real-time". PhD thesis. Université Joseph Fourier, 2008 (cit. on p. 53).

[Cer+05] E. Cerezo et al. "A survey on participating media rendering techniques". In: *Springer Verlag* 21.5 (June 2005), pp. 303–328. DOI: 10.1007/s00371-005-0287-1 (cit. on p. 135).

[DCH88] R. Drebin, L. Carpenter, and P. Hanrahan. "Volume rendering". In: *SIGGRAPH '88*. Vol. 22. 4. 1988, pp. 65–74. DOI: 10.1145/378456.378484 (cit. on p. 53).

[Del+10] C. Delalandre et al. "Single scattering in heterogenous participating media". In: *SIGGRAPH '10*. 2010, p. 1. DOI: 10.1145/1837026.1837044 (cit. on p. 135).

[Die04] T. Dieker. "Simulation of fractional Brownian motion". MA thesis. Department of Mathematical Sciences, University of Twente, 2004 (cit. on p. 58).

[Dob+00] Y. Dobashi et al. "A Simple, Efficient Method for Realistic Animation of Clouds". In: *SIGGRAPH '00 Proceedings*. 2000, pp. 19–28. DOI: 10.1145/344779.344795 (cit. on pp. 53, 107, 136).

[Dob+99] Y. Dobashi et al. "Using metaballs to modeling and animate clouds from satellite images". In: *The Visual Computer* 15.9 (Dic 1999), pp. 471–482. DOI: 10.1007/s003710050193 (cit. on p. 90).

[Dor+03] S. Dormido et al. *Procesamiento paralelo: teoría y programación*. Sanz y Torres, 2003 (cit. on pp. 24, 25).

[Ebe97] D. Ebert. "Volumetric modeling with implicit functions (A cloud is born)". In: *SIGGRAPH '97 ACM SIGGRAPH 97 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '97* (Jan. 1997), p. 147. DOI: 10.1145/259081.259233 (cit. on p. 52).

[Ele+12] O. Elek et al. "Interactive cloud rendering using temporally coherent photon mapping". In: *Computer & Graphics* 36.8 (Dec. 2012), pp. 1109–1118. DOI: 10.1016/j.cag.2012.10.002 (cit. on pp. 53, 136).

[Eng07] W. Engel. *Shader X5: Advanced Rendering Techniques (Shaderx)*. Charles River Media, 2007 (cit. on p. 52).

[ES00] P. Elinas and W. Stuerzlinger. "Real-time rendering of 3D Clouds". In: *Journal of Graphics Tools* (Feb. 2000), pp. 33–45. DOI: 10.1080/10867651.2000.10487531 (cit. on p. 63).

[Gar85] Y. Geoffrey Gardner. "Visual Simulation of Clouds". In: *Proc. ACM SIGGRAPH Computer Graphics*. Vol. 19. 1985, pp. 297–304. DOI: 10.1145/325334.325248 (cit. on pp. 50, 63).

[Gil88] B. Gil. *Mathematics I*. Edelvives, 1988 (cit. on pp. 12, 65).

[Gir12] Girishbalakrishnan. *Beyond Ink*. 2012. URL: https://wearebeyondink.wordpress.com/author/girishbalakrishnan/page/2/ (cit. on p. 52).

[GMF09] P. Gautron, J.E. Marvie, and G. Francois. "Volumetric Shadow Mapping". In: *SIGGRAPH '09*. 49. 2009. DOI: 10.1145/1597990.1598039 (cit. on p. 135).

[GN17] P. Goswami and F. Neyret. "Real-time Landscape-size Convective Clouds Simulation and Rendering". In: *Workshop on Virtual Reality Interaction and Physical Simulation*. Ed. by J. Fabrice and Z. Florence. The Eurographics Association, 2017. ISBN: 978-3-03868-032-1. DOI: 10.2312/vriphys.20171078 (cit. on p. 53).

[Häc06] H. Häckel. *Clouds: Identification guide*. Editorial Omega, 2006 (cit. on pp. 33, 36, 89).

[Har+05] M.J. Harris et al. "Simulation of cloud dynamics on graphics hardware". In: *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2005, p. 9. DOI: 10.1145/1198555.1198793 (cit. on pp. 2, 49, 51).

[Har02] M.J. Harris. “Real-time Cloud Rendering for Games”. In: *Proc. Game Developers Conference*. 2002, pp. 1–14 (cit. on pp. 51, 135).

[Har03] M.J. Harris. “Real-Time Cloud Simluation and Rendering”. PhD thesis. Chapel Hill, 2003 (cit. on pp. 41, 45, 51, 55, 56, 98).

[HB04] D. Hearn and M. P. et al. Baker. *Computer graphics with OpenGL*. Upper Saddle River, NJ: Pearson Prentice Hall, 2004 (cit. on p. 109).

[HL01] M.J. Harris and A. Lastra. “Real-time Cloud Rendering”. In: *Proc. Computer Graphics Forum (Eurographics 2001 Proceedings)*. Vol. 20. 2001, pp. 76–85. DOI: 10.1111/1467-8659.00500 (cit. on pp. 51, 69, 72, 136).

[Hor+05] L. Horng-Shyang et al. “Fast rendering of dynamic clouds”. In: *Computer & Graphics* 29.1 (Feb. 2005), pp. 29–40. DOI: 10.1016/j.cag.2004.11.005 (cit. on p. 135).

[HSA05] M. M. Hassan, K. M. Sazzad, and E. Ahmed. “Generating and Rendering Procedural Clouds in Real Time on Programmable 3D Graphics Hardware”. In: *9th International Multitopic Conference, IEEE INMIC 2005*. 2005, pp. 1–6. DOI: 10.1109/INMIC.2005.334442 (cit. on p. 61).

[Hua+08] B. Huang et al. “Study and Implement About Rendering of Clouds in Flight Simulation”. In: *Proc. IEEE '08*. 2008, pp. 1250–1254. DOI: 10.1109/ICALIP.2008.4590228 (cit. on pp. 2, 49, 51, 81, 135, 136).

[Iki+04] M. Ikits et al. *GPU Gems*. Addison-Wesley, 2004 (cit. on pp. 15, 16).

[Jac08] M. Jacobs. *Stark Trek Technology for Java3D*. 2008. URL: http://mikejacobs.ulitzer.com/node/99792/mobile (cit. on p. 51).

[Jar+08] W. Jarosz et al. “Radiance Caching for Participating Media”. In: *ACM Transactions on Graphics (TOG)* 27.1 (Mar. 2008). DOI: 10.1145/1330511.1330518 (cit. on p. 135).

[JG18] C. Jiménez de Parga and S.R. Gómez Palomo. “Efficient Algorithms for Real-Time GPU Volumetric Cloud Rendering with Enhanced Geometry”. In: *Symmetry* 10.4 (2018), p. 125 (cit. on pp. 55, 56, 111).

[JG19] C. Jiménez de Parga and S.R. Gómez Palomo. "Parallel Algorithms for Real-Time GPGPU Volumetric Cloud Dynamics and Morphing." In: *Journal of Applied Computer Science & Mathematics* 13.27 (2019) (cit. on pp. 98, 100, 106, 110).

[Jim03] C. Jiménez de Parga. *DirectMIDI homepage. Contributions*. 2003. URL: http://directmidi.sourceforge.net/ (cit. on p. 50).

[Jon01] H. Jones. *Computer Graphics through Key Mathematics*. Springer, 2001 (cit. on p. 87).

[Kal+17] S. Kallweit et al. "Deep Scattering: Rendering Atmospheric Clouds with Radiance-Predicting Neural Networks". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2017)* (Sept. 2017). DOI: 10.1145/3130800.3130880 (cit. on pp. 2, 53, 55, 56, 135).

[Kni+02] J. Kniss et al. "Interactive translucent volume rendering and procedural modeling". In: *IEEE Visualization, 2002. VIS 2002.* IEEE. 2002, pp. 109–116 (cit. on pp. 52, 55, 56).

[Knu98] D. Knuth. *The Art of Computer Programming 3. Sorting and Searching, 2. Ausgabe*. 1998 (cit. on p. 63).

[KPK15] S.Y. Kang, K.C. Park, and K.I Kim. "Real-Time Cloud Modeling and Rendering Approach Based on L-system for Flight Simulation". In: *International Journal of Multimedia and Ubiquitous Engineering* 10.6 (June 2015), pp. 395–406 (cit. on pp. 87, 135).

[KSE14] O. Klehm, H. Seidel, and E. Eisemann. "Prefiltered Single Scattering". In: *ACM I3D Proceedings* (Mar. 2014), pp. 71–78. DOI: 10.1145/2556700.2556704 (cit. on pp. 53, 136).

[Lav17] P. Laven. *MiePlot*. 2017. URL: http://www.philiplaven.com/mieplot.htm (cit. on p. 43).

[Lev88] M. Levoy. "Display of surfaces from volume data". In: *IEEE Computer Graphics and Applications* 8.3 (May 1988), pp. 29–37. DOI: 10.1109/38.511 (cit. on p. 53).

[LG05] B. Lipuš and N. Guid. "A new implicit blending technique for volumetric modelling". In: *Visual Computer* (Feb. 2005), pp. 83–91. DOI: 10.1007/s00371-004-0272-0 (cit. on p. 74).

[LWS98] S. Lee, G. Wolberg, and S.Y. Shin. "Polymorph: Morphing among multiple images". In: *IEEE Computer Graphics and Applications* 18.1 (1998), pp. 58–71 (cit. on p. 108).

[Max94] N. Max. "Computer Animation of Clouds". In: *Proc. 94,Geneva (Switzerland)*. 1994, pp. 167–174, 204. DOI: 10.1109/CA.1994.323994 (cit. on p. 50).

[Max95] N. Max. "Optical Models for Direct Volume Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (June 1995), pp. 99–108. DOI: 10.1109/2945.468400 (cit. on pp. 45, 51, 69).

[MB15] K. Mukina and A. Bezgodov. "The Method for Real-time Cloud Rendering". In: *Procedia Computer Science*. Vol. 66. 2015, pp. 697–704. DOI: 10.1016/j.procs.2015.11.079 (cit. on pp. 2, 50, 55, 56, 136).

[Mea80] D. Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic, 1980 (cit. on p. 17).

[Miy+01] R. Miyazaki et al. "A method for modeling clouds based on atmospheric fluid dynamics". In: *Proc. Computer Graphics and Applications, 2001. Ninth Pacific Conference in Tokyo*. 2001, pp. 363–372. DOI: 10.1109/PCCGA.2001.962893 (cit. on p. 49).

[Mon+17] A. Montenegro et al. "A new method for modeling clouds combining procedural and implicit models". In: *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE. 2017, pp. 173–182 (cit. on pp. 135, 136).

[MR05] P. Matt and F. Randima. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Educational Publishers, 2005 (cit. on p. 117).

[MV68] B.B. Mandelbrot and J.W. Van Ness. "Fractional Brownian motions, fractional noises and applications". In: *SIAM review* 10.4 (1968), pp. 422–437 (cit. on p. 58).

[Nar+06] S.G. Narasimhan et al. "Acquiring Scattering Properties of Participating Media by Dilution". In: *ACM Transactions on Graphics (TOG)* 25.3 (July 2006), pp. 1003–1012. DOI: 10.1145/1141911.1141986 (cit. on p. 135).

[nVi12] nVidia. *nVidia CUDA C Programming Guide*. 2012. URL: http://docs.nvidia.com/cuda/-cuda-c-programming-guide/index.html (cit. on pp. 29, 30).

[Paw97] J. Pawasauskas. "Volume Visualization With Ray Casting". In: *CS563 - Advanced Topics in Computer Graphics Proceedings*. Feb. 1997 (cit. on pp. 16, 18).

[Per85] K. Perlin. "An image synthesizer". In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296 (cit. on p. 58).

[Pet+16] C. Peters et al. "Beyond hard shadows: moment shadow maps for single scattering, soft shadows and translucent occluders". In: *ACM I3D Proceedings* (Mar. 2016), pp. 159–170. DOI: 10.1145/2856400.2856402 (cit. on pp. 53, 136).

[PL96] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Berlag, 1996 (cit. on pp. 3, 87).

[Qui15] I. Quilez. *Clouds*. 2015. URL: https://www.shadertoy.com/view/XslGRr (cit. on p. 108).

[Qui18] I. Quilez. *Advanced Value Noise*. 2018. URL: http://www.iquilezles.org/www/articles/morenoise/morenoise.htm (cit. on p. 58).

[Ree83] W. T. Reeves. "Particle Systems - a Technique for Modeling a Class of Fuzzy Objects". In: *ACM Trans. Graph.* 2.2 (Apr. 1983), pp. 91–108. ISSN: 0730-0301. DOI: 10.1145/357318.357320. URL: http://doi.acm.org/10.1145/357318.357320 (cit. on p. 51).

[Rod13] J. Rodriguez Villar. *GLSL Essentials*. Packt Publishing, 2013 (cit. on pp. 20–22).

[RY96] R. Rogers and M. Yau. *A Short Course in Cloud Physics. Third Edition*. Butterworth-Heinemann, 1996 (cit. on pp. 36, 38, 39).

[SBN94] M.A. Sebastián, V. Barquero, and V. Novo. *Gestión y Control de la calidad*. Madrid-UNED, 1994 (cit. on p. 9).

[SK03] P. Shirley and R. Keith. *Realistic Ray-Tracing, Second Edition*. A K Peters, 2003 (cit. on pp. 14, 63).

[SK10] J. Sanders and E. Kandrot. *CUDA by example. An introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010 (cit. on p. 29).

[Smi02] B. Smits. "Efficient bounding box intersection". In: *Ray Tracing News* 15.1 (Oct. 2002), p. 1 (cit. on pp. 3, 66).

[Sta03] J. Stam. "Real-time fluid dynamics for games". In: *Proceedings of the game developer conference*. Vol. 18. 2003, p. 25 (cit. on pp. 98–100).

[Tes16] J. Tessendorf. *Resolution Indepent Volumes*. School of Computing, Clemson University, 2016 (cit. on pp. 69, 72).

[WBC08] J. Wither, A. Bouthors, and M.P. Can. "Rapid sketch modeling of clouds". In: *Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM)* (June 2008), pp. 113–118 (cit. on p. 136).

[Wei95] M.A. Weiss. *Data Structures and Algorithm Analysis (2Nd Ed.)* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1995. ISBN: 0-8053-9057-X (cit. on p. 63).

[Whi79] T. Whitted. "An improved illumination model for shaded display". In: *Proc. SIGGRAPH '79*. Vol. 13. 1979, p. 14. DOI: 10.1145/965103.807419 (cit. on pp. 2, 12).

[Wil+05] A. Williams et al. "An efficient and robust ray-box intersection algorithm". In: *SIGGRAPH '05 ACM SIGGRAPH 2005*. 9. 2005, pp. 1–4. DOI: 10.1145/1198555.1198748 (cit. on p. 66).

[Yus14] E. Yusov. "Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics". In: *High-Performance Rendering of Realistic Cumulus Clouds Using Pre-computed Lighting*. 2014, pp. 127–136. DOI: 10.2312/hpg.20141101 (cit. on pp. 53, 135).

[YW11] C.M. Yu and C.M. Wang. "An Effective Framework for Cloud Modeling, Rendering, and Morphing." In: *J. Inf. Sci. Eng.* 27.3 (2011), pp. 891–913 (cit. on pp. 108, 136).

[Zho+08] K. Zhou et al. "Real-time smoke rendering using compensated ray marching". In: *SIGGRAPH '08*. 36. 2008, pp. 1–12. DOI: 10.1145/1399504.1360635 (cit. on p. 135).

Analytical index

A

Absorbance . . . 41
Absorption . . . 40
Adiabatic . . . 37, 38
Adiathermal . . . 37
Advection . . . 46, 54, 57, 97, 99, 102
Aerosols . . . 33
Air . . . 33, 38, 39
Air layers . . . 33
Al-Hazen . . . 40
Albedo . . . 41
Alpha test . . . 22
Altocumulus . . . 35
Altostratus . . . 35
Amdahl's law . . . 24
Appel . . . 2, 12
Aristid Lindenmayer . . . 3, 87
Artificial intelligence (AI) . . . 50
Asymmetry factor . . . 73
Atmosphere . . . 33, 36
Average case . . . 116

B

Backus-Naur form (BNF) . . . 88
Barycenter . . . 91, 110
Basic input output system (BIOS) . . . 18
Benchmarks . . . 115
Best case . . . 116
Big data . . . 23
Black-box testing . . . 9
Blender . . . 8, 90
Blending . . . 22
Blinn . . . i, 16, 90
Block . . . 30, 76
Bounding box . . . 63
Bouthors . . . 136
Buoyant force . . . 38
Bus . . . 18

C

C++ . . . 4, 31, 86, 88, 102, 104, 119
C++ warnings . . . 9
C-based programming . . . 22
Carbon dioxide . . . 36
Castellanus . . . 89
Chip . . . 18
Cirrocumulus . . . 34
Cirrostratus . . . 34
Cirrus . . . 34
Clausius-Clapeyron equation . . . 39
Climate change . . . 23
Clusters . . . 25
Complexity analysis . . . 115
Computed tomography . . . 13
Computer games . . . i, 49, 53, 65, 89, 138
Computer graphics . 2, 4, 17, 27, 49, 53, 54, 65, 97, 114
Congreso Español de Informática Gráfica . . 8
Conjugate gradient . . . 97
Contributions . . . 7

Convection 46, 97
Cornell University 88
CPU . . . 2, 7, 18, 19, 23, 27, 28, 31, 70, 74, 93, 112, 117, 130
CUDA . . . 11, 27–31, 74–76, 100, 102–104, 106, 115, 119, 129, 130, 136
Cumulonimbus 36
Cumulus 35

D

Deformation 108
Depth test 22
Descartes 40
Deterministic 20
Dew point 33
Diffusion 103
Digital art 89
Digital-to-analogue converter (RAMDAC) 18
Direct3D 9 20
DirectX 3, 20
Dobashi 90
Droplet 33
Dry air 36, 38, 39

E

Educational software i
Electronic signal 18
Emitter 51
Ethernet LAN 27
Euclidean 3, 12, 62, 63, 65, 66, 95
Euler 99
Extinction 41

F

FBm 58, 60, 61, 90, 137
Filter 83
Financial and economic modelling 23
First Law of Thermodynamics 37
Flight simulators . . . . . . i, 138
Fluid dynamics 46
Fluid engine 130
Flynn taxonomy 25
Foucault 40
FPS 9, 66, 116, 133, 136
Fractal Brownian motion (fBm) noise . . . 57
Fragment shader 21, 66, 67
Framebuffer 22
Free atmosphere 33
Fresnel 40

G

Gamma function 58
Gardner . . . . . . i, 50, 63
Gauss-Seidel 97, 100
Gaussian 51, 52, 60, 61, 81–83, 137
GeForce 1030 GT 117, 119
GeForce 8800 GTS 117
GeForce GTX 1050 non-Ti 117, 119
GeForce GTX 1070 non-Ti 119
GeForce GTX 970 117
Geometry shader 22
GLM 117, 119
Global memory 29
GLSL 4, 8, 21, 22, 66, 88, 96, 110, 112, 119
Goethe 34
Gouraud 52
GPGPU 23
GPU 2, 6, 7, 11, 12, 18, 20, 23, 27, 28, 30, 50, 56, 57, 64, 66, 67, 70, 74, 75, 90, 93, 115, 116
Gravity 38, 51
Greedy heuristic 74
Grid . 13, 25, 29, 48, 58, 66, 69, 71, 73, 76, 102
Guide points 106
Gustav Mie 43

H

Heat . . . 40
Helium . . . 36
Henyey-Greenstein phase function . . . 43
Hurst parameter . . . 60
Huygens . . . 40
Hydrostatic equilibrium . . . 38
Hypertexture . . . 7, 53, 57, 58, 117
Hypothesis . . . 4

I

Ideal gas . . . 36
Insertion Sort . . . 63, 116
Intel Xeon . . . 135
ISO . . . 9

J

Jacobi . . . 97, 100, 103
Java . . . 9
Jean Baptiste Lamarck . . . 34
Jesse L. Greenstein . . . 43
Jet streams . . . 89
John William Strutt . . . 42
Journals . . . 8

K

K-D trees . . . 66
Kajiya . . . i, 16
Kernel . . . 28, 29, 75, 103, 104
Keyframe . . . 109
Kruskal-Wallis . . . 86

L

L-system . . . 3, 7, 9, 54, 87
Laplacian operator . . . 100
Lebesgue-Stieltjes integral . . . 59
Level of condensation . . . 87
Level of detail . . . 7
Lifecycle . . . 4
Light beams . . . 138
Local memory . . . 29
LOD . . . 65
Louis G. Henyey . . . 43
Luke Howard . . . 34
Lumion . . . 50, 137

M

Magnetic resonance imaging . . . 13
Mark Harris . . . 45
MATLAB . . . 8, 58, 74
Maximum likelihood estimators . . . 84
Medical visualization . . . 13
Medicine . . . 23
Megaparticles . . . 52, 54
Memory leaks . . . 9
Metaballs . . . 90, 117, 119
Metamorphosis . . . 109
Meteorology . . . 33
MiePlot . . . 43
Moist air . . . 36, 39
Moore's law . . . 25, 49
Morphing . . . 111, 136
Motorola 68000 . . . 25
Multicore . . . 4, 23, 25
Multiple scattering . . . 41

N

Navier-Stokes . . . 1, 7, 11, 46, 97, 98, 138
NDT . . . 71, 74–76, 117
Nelson Max . . . 45
Neon . . . 36
Neural network . . . 2, 135
Neurobiology . . . 88
Newton . . . 40, 97

Nimbostratus ... 36
Nimbus framework ... 8
Nitrogen ... 36
No-duplicate-tracing (NDT) algorithm ... 69
Nondeterministic ... 29
NVidia Titan ... 135

O

Object-oriented ... 9
Octree ... 17, 66
Ontogenetic ... i, 49, 54, 108
OpenGL ... 3, 4, 20, 88, 107, 117, 119
OpenGL 1.x ... 20
Optical depth ... 41
Optimization ... 119
Oxygen ... 36
Ozone ... 36

P

Parallel computing ... 25
Parallel programming ... 23
Pareidolic ... 96, 138
Perlin ... 58, 60
Phase functions ... 42
Phong ... 52
Photon ... 41
Pipeline ... 16, 20, 22, 23
Pointers ... 103
Poisson ... 100
Polarisations ... 43
POV-Ray ... 137
Precipitation ... 138
Precomputing phase ... 76
Przemyslay Prusinkiewicz ... 87
Pseudoellipsoid ... 95, 110
Pseudosphere ... 7, 57, 61, 74, 106, 108, 137
Pseudospheroid ... 2, 54, 61, 63, 67, 69, 70, 81, 82, 115, 137
PTX ... 32

Q

Q-Q diagrams ... 85
Quadtrees ... 66
Quality ... 9
Quicksort ... 116

R

R statistical application ... 84
Radiometry ... 40
Ray-collision ... 12
Raycasting ... 11–13, 16, 17
Raymarching 12, 13, 57, 61, 66, 69, 71, 72, 74, 90, 115, 117, 119, 136, 137
Raytracing ... 2, 11–13, 15, 17, 57, 79, 135
Realism ... 7
Reflection ... 72
Rodrigues' Rotation Formula ... 91
Runge-Kutta ... 99

S

Scattering ... 71, 72
Scattering angle ... 73
Scissor test ... 22
Shaders ... 20, 22
Shadow mapping ... 138
Shared memory ... 29
Silver lining ... 78
Single Instruction Multiple Thread (SIMT) ... 31
Single scattering ... 41
Skybox ... 50, 54
Skydome ... 50, 54
Smits ... 3, 66
Snell ... 40
Software ... 8
Specific objectives ... 4

Speedup 24, 115, 130, 136
Spiral 4
Stam 100
Steradians 72
Stratocumulus 35
Stratus 35
Streaming multiprocessors 30
Sun/Oracle 9
Superlinear speedup 24
Syncthreads() 28

T

Temperature 33, 38, 39
Thermal inversion 33
Threads 28–30
Transmittance 71, 72
Trilinear interpolation 73

U

UN 34
User-friendly 4

V

Vertex shaders 21
Video graphics array (VGA) 18
Video random access memory (VRAM) . . . 18
Videogames 27
Virtual reality 1, 65, 137, 138
Viscosity 100
Visual C++ 8, 119
Volumetric rendering 2, 52, 53
Voxel . 13, 66, 70, 71, 73–76, 87, 100, 112, 137

W

Warps 30
Water 33, 44
Water vapour 33, 36, 39, 40
WebGL 108
White-box testing 9
Whitted 2, 12
Wind 51, 106
Wireframe 93, 110
World Meteorological Organization 34
Worst case 116

Y

Young 40

Copyright notice

- LaTeX modified template is CC BY 4.0 by Vincent Labatut.
- Cover images are royalty-free by Alex Landa and Inga Nielsen.
- Figure 2.5 is CC BY-SA 3.0 by WhiteTimberwolf.
- Figure 2.11 is CC BY-SA 3.0 by Daniels220.
- Figure 2.25 is CC BY-SA 3.0 by Valentin de Bruyn / Coton.
- Galaxy morphology is CC BY-NC-SA 3.0 by S. Guillitte.
- Non physical based atmospheric scattering is CC BY-NC-SA 3.0 by robobo1221.
- Elevated is CC BY-NC-SA by Iñigo Quilez.
- Black moon is CC BY-NC-SA by DeMaCia.
- Seascape is CC BY-NC-SA by Alexander Alekseev aka TDM.