Computer Science • 18(3) 2017 http://dx.doi.org/10.7494/csci.2017.18.3.1491

Paweł Kobak
Witold Alda

MODELING AND RENDERING OF CONVECTIVE CUMULUS CLOUDS FOR REAL-TIME GRAPHICS PURPOSES

Abstract *This paper presents a simulation and rendering model of three-dimensional convective cloud evolution. The model is physically based; however, its purpose is graphical. The main stress is put on balancing two parts of the model: the atmosphere simulation with the convective motion of air and water vapor combined with the rendering of semi-transparent and light-scattering clouds in order to achieve realistic animation in real-time. We examine and compare two algorithmic approaches based on CPU and GPU computations.*

Keywords computer graphics, natural phenomena, GPU programming

Citation Computer Science 18(3) 2017: 241–268

1. Introduction

Computer graphics is a rapidly growing field with numerous applications, such as video games, movies, visualization tools, and various simulators. In most cases in which the scene presents open air and the sky, it is important to keep a high level of realism in cloud modeling and rendering. A common way to obtain good image quality is the use of cloudy-sky images created in advance by using graphics tools or photos. The disadvantage of this approach is that the obtained image is static and non-interactive.

In order to obtain a dynamic image of clouds, one can use methods that generate random cloud-like images. These methods are not based in any way on a realistic model of the atmosphere, but they can produce a decent dynamic image of the sky. Using a realistic simulation of the atmosphere would provide better results, but this task is beyond the capacity of an ordinary computer, especially if the image must be generated in real-time. There are, however, simplified models of the atmosphere that are much easier to compute. They give approximate results and do not include all of the processes, yet they seem to be sufficient in meeting the needs of computer graphics.

The aim of the study is to examine the possibility of using the resources of a modern personal computer for the generation of smooth real-time animation of a scene with realistic time-varying clouds. The scope of the work includes the implementation of a simplified model of atmosphere simulation that produces data for the modeling of clouds as well as a rendering algorithm that is attached to it. Our computations have been performed on both CPU and GPU processors.

The range of this model is limited to clouds resulting from the process of convection. Due to the local nature of this phenomenon, it is possible to build relatively simple models that can be calculated in real time.

2. Clouds – classification, and the process of formation

Clouds are composed of water droplets or ice crystals and are produced by the condensation of water vapor existing in the atmosphere.

Water vapor condenses when its volume in the air exceeds the saturation level. Pressure of saturated vapor increases exponentially when temperatures increase. Thus, warmer air can contain much more water vapor than cold air.

Increasing the amount of water vapor in the atmosphere or cooling the wet air may, thus, lead to the formation of clouds.

2.1. Cloud classification

Clouds can be divided into several categories based on their appearance, method of creation, and height at which they appear [15]. A brief classification below contains the main cloud categories divided into strata.

High stratum clouds.

- Cirrus clouds, occurring at high altitude (over 6 km), are composed of ice crystals with a delicate fibrous appearance.
- Cirrostratus clouds, at a height of 6–12 km, are in the form of a transparent, often almost invisible veil.
- Cirrocumulus clouds, small puffy clouds occurring at high altitude. The smallest of the family are small enough that they do not cast shadows.

Medium stratum clouds.

- Altocumulus clouds consist primarily of water droplets, often in the form of small clouds or belts. Larger than cirrocumulus clouds, they are distinguishable by the occurrence of shadows.
- Altostratus cloud. Similar to cirrostratus but at a lower altitude and less transparent – seen as a dense layer through which the Sun is barely visible.

Low stratum clouds.

- Nimbostratus, dark clouds whose base is mostly below 2 km, made up of water droplets.
- Stratocumulus, cumulus clouds occurring at low altitudes, significantly larger than altocumulus. They may be dark or light gray with shapes that enable us to see patches of clear sky.

Vertically developed clouds.

- Cumulus Clouds, thick white clouds with flat bases. Formed as a result of convection, they are powered by the warm moist air floating up from the surface of the Earth.
- Cumulonimbus clouds, rainy, much larger than cumulus, may be several kilometers high. Accompanied by strong air currents and storms.

In this work, we will concentrate only on the final group from the list above.

2.2. Convection

One of the processes leading to the formation of clouds is convection. The surface of the Earth is heated by the Sun, which in turn heats the air. Warmer air rises upward, where it mixes with cooler air. As a result of this process, convection cells are formed; i.e., warm rising air current is accompanied by falling cooler air current, which additionally causes a horizontal movement of air at the base and top of the vertical currents. If water evaporation at the base of the current causes a subsequent inflow of steam, this will rise to the higher parts of the atmosphere, where the lower air temperature will cause the condensation of vapor to droplets of water or ice crystal suspension, eventually forming clouds.

3. Methods for cloud modeling

There have been numerous attempts to model the evolution of clouds for computer graphics needs. Many of these use the procedural methods – clouds are generated randomly, as in the case of Perlin noise [11, 12]. There are also models that use simplified models of atmospheric convection.

According to the way clouds are represented and to the atmospheric state, the models can be divided into two groups: those that simulate the formation of clouds using spherical particles representing an element of a cloud (or steam), and those that use three-dimensional mesh to keep the state of the atmosphere.

3.1. Particle-based models

In the models based on particles used by Neyret [8] and Bouthors [1], for example, the simulations use spherical air bubbles that can interact with each other. Near the Earth's surface, warm air bubbles are generated. These float upwards; after reaching the appropriate height at which water vapor condensation takes place, they transform into clouds.

The fine details of cloud shapes are modeled by generating bubbles with smaller radii and placing them on the surface of larger bubbles located at the edge of the cloud. The calculation cost of particle models is dependent on the number of active particles. This means that we can expect a slowdown in the simulation with increased cloudiness.

3.2. Mesh-based models

Mesh-based models usually exploit a regular three-dimensional grid of equally spaced points containing model data such as pressure, temperature, humidity, etc. Computational cost of such a model does not depend on the state - this is constant, regardless if we are dealing with a cloudless sky or one completely full of clouds.

The simplest version of the mesh model introduced by Nagel [7] is based on a cellular automaton. It describes the state of the atmosphere at the grid node with three logical values specifying the presence of water vapor, clouds, and phase transition. The values of water vapor and phase transition are initialized randomly, while further evolution of the clouds is controlled by cellular automaton rules.

In his version of the model, Dobashi [2] develops a Nagel method by adding the ability of disappearance and re-appearance of the clouds according to random distribution. Additionally, water vapor can be added at random (to initiate phase transitions) as well as wind (which can move clouds horizontally).

The simulation of cloud evolution implemented in our paper (which is also a mesh model) is a modified version solution described by Miyazaki [6]. Miyazaki uses a Coupled Lattice Model (CML) originally developed by Kaneko [4]. CML may be treated as an extension of a cellular automaton. The idea behind it is the usage of recurrence equations applied to a regular lattice. The model is widely used to describe a number of dynamic systems, especially to demonstrate spatial chaotic phenomena in phase transitions, chemical reactions, and others. In our model, bit values of the atmosphere parameters are replaced by floating-point numbers, while cellular automaton rules are replaced by continuous functions. This allows us to better describe the gradual changes in the atmosphere.

The model is more complicated – we introduce air speed and temperature. The change of the model state is influenced by such phenomena as diffusion, buoyancy, phase transition dependent on temperature, and water vapor content, as well as the effects of pressure affecting air velocity. Convection is modeled by increasing the temperature and humidity as well as introducing vertical air currents.

We get a simplified yet realistic simulation model of phenomena in the atmosphere that lead to the formation of convective cumulus clouds. Among its advantages are achieving high-quality realistic-looking results while maintaining relatively simple rules; this allows the simulation to be performed in real time on a sufficiently large mesh so that the results can be used to render a cloudy sky.

4. The model of the atmosphere

This section provides a detailed description of the simulation model, the way the state of the atmosphere is represented, and which phenomena are taken under consideration. We will explain how they affect the changes in the atmosphere.

The implementation of the simulation model of the atmosphere is a modified version of the model described by Miyazaki [6].

The model is relatively simple computationally and has a linear complexity along the number of grid nodes – for each node, a constant number of operations is performed. Since the operations for each node are the same and are local in all interactions, the model is well-suited for parallelization.

4.1. Computing mesh

Calculations are performed on a three-dimensional mesh of equally spaced points in space. For each node, its state is described by six floating point numbers corresponding to four parameters of the atmosphere:

- Air temperature T. The preset value is defined as the difference of the current temperature and the base temperature at a given height, which means that, at the beginning of the simulation, $T = 0$ is set everywhere. The base temperature is set as the temperature at ground level modified by the temperature decline per unit height.
- Humidity w_v.
- Number of water droplets w.
- Velocity vector $\vec{v} = [v_x, v_y, v_z]$ describing the direction and speed of the movement of air masses. For the sake of simulation stability, it is required that the absolute value of velocity vector components does not exceed 1, which corresponds to displacement by a distance equal to the distance of adjacent grid cells.

The boundary conditions are periodic for horizontal directions (x and y) and closed-absorbing conditions for vertical direction (z).

4.2. Included phenomena

4.2.1. Diffusion

The simulation uses a discrete diffusion model wherein a new value depends on the difference values in a current, and the values in the six neighboring nodes spaced by 1 in each direction on x, y, z. Diffusion parameter P is calculated according to the following formula:

$$P(x,y,z)^* = P(x,y,z) + K_{dP} * \Delta P(x,y,z)$$

where K_{dP} is the parameter controlling the force effect, while $\Delta P(x,y,z)$ is:

$$\frac{1}{6}\Big(P(x+1,y,z) + P(x-1,y,z) + P(x,y+1,z) + P(x,y-1,z) \\ + P(x,y,z+1) + P(x,y,z-1)\Big) - P(x,y,z)$$

The parameters for which diffusion is considered are water vapor, water droplets, temperature, and air speed (separately for each component).

4.2.2. Pressure

The role of pressure is to equalize the quantity of air entering and exiting from the node of the mesh. For x, the component of the formula is as follows:

$$\begin{aligned} g_x(x,y,z) = &\frac{1}{2}\Big(v_x(x+1,y,z) + v_x(x-1,y,z) - 2v_x(x,y,z)\Big) \\ &+ \frac{1}{4}\Big[v_y(x+1,y+1,z) + v_y(x-1,y-1,z) - v_y(x-1,y,z+1) \\ &- v_y(x+1,y-1,z) + v_z(x+1,y,z+1) + v_z(x-1,y,z-1) \\ &- v_z(x-1,y,z+1) - v_z(x+1,yz-1)\Big] \end{aligned}$$

Values for the components of y and z are calculated in a similar way.

If more air flows in than out in any of the neighboring cells, the velocity component that generates the flow of air out is increased and vice versa, thus reducing the amount of the flow to zero in average.

The complete formula for the new speed value in the next step (taking the viscosity and pressure into account) is as follows:

$$v_i^{new}(x,y,z) = v_i(x,y,z) + k_{dv}\Delta v_i(x,y,z) + k_p g_i(x,y,z),$$

for $i \in (x,y,z)$, where k_{dv} is the viscosity coefficient and k_p is the pressure-effect coefficient.

4.2.3. Buoyancy

The buoyancy effect modifies the vertical component of v_z velocity in the current node according to the difference between itself and four neighboring nodes in the directions of x and y. The model implemented in the standard version provides a buoyancy force facing up to the nodes of a higher temperature than its neighbors. Additionally, we have also included the buoyancy force introduced by Harris [3]; this is directed upward depending on the amount of water vapor and facing down depending on the rate of the water droplets. The numerical scheme is as follows:

$$v_z^{new}(x,y,z) = v_z(x,y,z) + \frac{K_{bt}}{4}(v_z(x-1,y,z) + v_z(x+1,y,z) \\ + v_z(x,y-1,z) + v_z(x,y+1,z))$$

where K_{bt} is the factor that controls the force effect.

4.2.4. Advection

Advection is responsible for the displacement of the values stored in a mesh node to a new location due to air movement. The cell parameters that are subject to advection are the amount of water vapor, number of water droplets, and temperature. Data from the point of $\vec{P} = (n_x, n_y, n_z)$ are transferred to the point of $\vec{P_n} = \vec{P} + \vec{V}(n_x, n_y, n_z)$. Subsequently, they are distributed between the eight vertices of the unit cube containing point $\vec{P_n}$. The cell parameters that are subject to advection are the amount of water vapor, number of water droplets, and temperature.

4.2.5. Heating Earth surface and water evaporation

To simulate the heating of air masses over a heated Earth surface, the temperature of the lowest layers of the mesh is increased by the value imported from two-dimensional temperature maps loaded from an image file. Bilinear interpolation is performed between the map elements.

Analogously, the same map is used to control the increase of the amount of water vapor in the lowest layer of the grid. Placing the source of vapor at heated points causes its advection together with rising air.

4.2.6. Phase transition

Phase transition refers to the process of vapor condensation and the inverse conversion process that changes water droplets into steam. The maximum amount of water vapor in the air, v_{max}, depends on temperature T and is described by the Clausius-Clapeyron formula in terms of vapor pressure, where pressure $p \sim exp(\frac{-1}{T})$. We follow an empirical formula from Miyazaki's work [6]:

$$v_{max}(T) = 217e^{(19.482 - \frac{4303.4}{T-29.5})}/T$$

If the amount of vapor in the cell is greater than the critical value, it will partly condense; otherwise, the reverse process occurs. The amount of water that will change the physical state of the simulation step depends on phase transition speed parameter α. The new values of water vapor v^* and water droplets w^* are as follows:

$$v^* = v - \alpha(v - v_{max})$$

$$in^* = w + \alpha(v - v_{max})$$

Temperature changes in the node due to the exo-/endo-thermic character of the phase transition are also taken into account:

$$T^* = T - Q(v - v_{max}(T))$$

Coefficient Q is a parameter specifying the amount of emitted or absorbed heat.

4.2.7. Wind effects

The effect of wind is the same for each cell and is used to allow for the movement of clouds. Large shifts breach a condition limiting velocity components to $[-1, 1]$ and cause problems with interpolation used during rendering, which results in "jumpy" motion instead of smooth motion. To prevent this effect, the wind has been implemented in the form of translation vector $\vec{W}i = (w_x, w_y)$ assigned to each step of the simulation.

This allows for the smooth animation of cloud shifts equal to a few distances between grid nodes in a single frame of the simulation.

The new offset value in step $i + 1$ is calculated by adding the current strength of the wind to step i. The current wind strength is calculated on the basis of the simulation input parameters (base strength and wind direction), which are subsequently distorted in a random fashion.

The formula for instantaneous wind vector for i in the frame simulation is as follows:

$$\vec{W}_i = (F_i \sin(\beta_i), F_i \cos(\beta_i))$$

Angle β_i and wind strength F_i are calculated on the basis of angle β_b and strength F_b as well as on the variation of wind direction (β_{var}) and its strength (F_{var}).

$$\beta_i = \beta_b + r_\beta \beta_{var}$$

$$F_i = F_b + r_f F_{var}$$

where r_β, r_f are random variables with normal distribution and variance equal to 1.

4.2.8. Turbulence

Stam [14] showed that presenting phenomena such as smoke, gas, and steam as well as the use of small-scale small random turbulence helps to achieve more realistic, less-regular shapes.

Due to performance reasons, the turbulence model implemented here is much simpler than that described by Stam. During the advection step, a velocity vector is added to a vector of length in range $[0, F_t]$ and random direction. F_t is the parameter controlling the strength of turbulence.

4.3. Input parameters

The following table 1 lists all of the parameters of the model along with a brief description of their meanings for the simulation.

Table 1
Parameters of the model

Parameter	Description
α	Phase transition speed
β_b	Base wind direction
β_{var}	Variability of wind direction
cs	Distance in meters between adjacent nodes of the computational grid
F_b	Base wind force
F_t	Turbulence strength
F_{var}	Variability of wind force
h_{low}	Height that is at the bottom level of computing box
k_{bt}	Scaling parameter for air buoyancy resulting from temperature
k_{bv}	Scaling parameter for air buoyancy resulting from mass of water vapor
k_b	Scaling parameter for air buoyancy resulting from mass of water droplets
k_{dt}	Scaling parameter for thermal diffusion
k_{dv}	Scaling parameter for water vapor diffusion
k_d	Scaling parameter for effect of condensed water diffusion
k_{dwv}	Scaling parameter for effect of thermal diffusion
n_t	Amount of energy added in a simulation step
t_v	Amount of steam added in a simulation step
k_p	Scaling parameter for pressure effect
T_0	Initial temperature on 0 height level (in Kelvin)
T_f	Temperature decline for every 100 m of altitude
s_x, s_y, s_z	Computational grid size
Q	Ratio of amount of energy absorbed/released during phase transition

5. Model implementation

The following section describes the implementation of a simulation in a multithreaded version running on a CPU as well as a GPU version using the Nvidia CUDA technology. The performance of the two implementations is compared.

In general, the simulator keeps the state of the grid in the set of arrays; one for each parameter in a grid cell. A second identical set of arrays is used to store temporary data when calculating the next step of the simulation.

The simulations use periodic boundary conditions in the x and y directions, and closed absorbing ones in the z direction. The coordinates have to be processed as follows:

$$\begin{aligned} x_n &= x \pmod{s_x} \\ y_n &= y \pmod{s_y} \\ z_n &= max(0, min(1, z)) \end{aligned}$$

A separate table is used to store the data for rendering purposes. Four sets of data (two for the density of the water droplets and two for the amount of light) are alternately stored as four-component RGBA color elements in texture memory so the program executed on the GPU can read them in a single operation.

5.1. Structure of the simulation step

In both implementations, the simulation of a step is divided into several sub-steps within which it is possible to parallelize the calculations. These are as follows:

The calculation of the new cell value includes the effects of diffusion, buoyancy, and phase transition as well as pressure and viscosity. Since these functions make use of the neighboring cells, the original content of the cell must remain unchanged, so the result is written to a temporary array.

5.1.1. Adding energy and humidity

This part is separated from the main part of the simulation for performance reasons. Operations contained therein have to be calculated only for the lowest layer of the grid. Since there are no references to other cells of the operation, it is done "in place" – the old values are overwritten by the new plus the new energy and water vapor.

5.1.2. Advection

The data from each cell is divided between the other cells depending on the velocity vector. The final value of the cell is the sum of several values from different locations. This makes parallelization more difficult and enforces the use of the additional synchronization mechanisms described in detail in the implementation section.

5.2. Simulation modes

The application allows for two modes of execution simulation:

- Simulation with maximum speed, when the next step is started immediately after the previous one has ended. In this mode, rendering is turned off.
- Simulation with a fixed-time interval defined by the user. It is used with rendering turned on. The rendering system interpolates the results from Steps i and $i+1$ for visualization while the simulation system calculates and stores Step $i+2$. After the defined time step, results from $i+2$ are moved to $i+1$ and the simulator starts the calculation of Step $i+3$. Setting smaller time intervals speeds up animation; however, if they are smaller than the time required to calculate the simulation step, the animation is no longer smooth.

5.3. CPU Implementation

The implementation on a CPU parallelizes calculations using a thread mechanism (with the help of the POSIX Threads library).

In the first two sub-steps of the simulation step, the computational mesh is divided into layers along x-coordinate, from 0 to s_x. Having k threads, current thread i performs calculations in layers $n \cdot k + i$ for $n = 0, 1, 2, ...$ until s_x is exceeded.

In the case of advection, calculations for cell in layer i may affect the values in layers $i - 1$ and $i + 1$. This interaction is limited to the nearest layers, provided the modulus of the velocity does not exceed 1. To prevent collisions in the calculations performed by different threads, advection is calculated in three stages: in the first one, layers $(n \cdot k + i)$ are taken into account, $(n \cdot k + i) + 1$ in the second, while $(n \cdot k + i) + 2$ is in the third and final step. Between subsequent stages, a thread synchronization is performed. Such an organization of the tasks provides that, between any layers calculated concurrently, there are always at least two layers for which no calculation is performed so that synchronization is not necessary at the level of the individual grid cells.

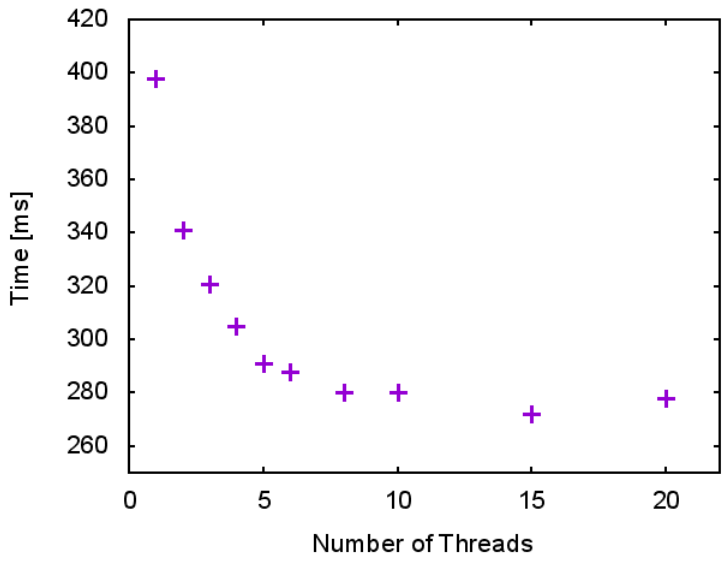

Figure 1. Time of a single frame calculation vs number of threads (averages over 100 frames are calculated).

Figure 1 shows the average time of the single frame calculation for the different number of threads. The test was performed on a dual-core processor 2.2GHz AMD Athlon X2.

5.4. GPU implementation

The increased computing capabilities of modern graphics cards, the opportunities they offer to parallelize computations, and the creation of dedicated environments for them to perform calculations for general-purpose arrays all have lead to an increase in the popularity of calculations performed on a GPU. The implemented simulation uses the Nvidia CUDA environment that runs on Nvidia graphics cards. The program code written in a language similar to C makes it easy to adjust the version intended for execution on a CPU.

A GPU consists of one or more multiprocessors that represent the SIMD (Single Instruction Multiple Data) architecture. Each of them can simultaneously execute the same instructions for hundreds of data sets (the exact number depends on the model of the graphics card). A GPU operates at a frequency lower than the average CPU; however, due to their massive parallelism, calculations on a GPU are almost always faster than on a CPU.

The parallelization of calculations is carried out by dividing them into threads grouped into blocks. At any given moment, the multiprocessor performs threads that belong to a single block. The scheduling of thread calculation is performed automatically by the CUDA environment.

The graphics processor architecture is best suited for performing a large number of identical threads. Differences in multiprocessor instructions executed for the various threads resulting from a conditional statement, for example, results in the fact that different options are executed sequentially, which significantly increases computation time.

The presented atmospheric simulation model perfectly meets the requirements of the algorithm to be well-suited to run on graphics processors. For each cell in a three-dimensional grid, the same calculations are performed (one cell per thread). Memory organization is the same as in the CPU implementation. The only addition element is an integer array that stores the state of the random number generator values for each cell separately. This is because all threads in an entire grid run simultaneously, and each of them must have a separate random number generator with its own state.

The simulation step is divided into the same sub-steps as in the CPU implementation. Calculation of new values in the cells (adding heat, humidity, and advection) can be performed using a single CUDA kernel call per step. In the case of advection, the parallel execution of threads can cause collisions to occur when the displaced values are added in their new locations. This is resolved through the atomic incrementation available on graphic processors that support a computing capability 2.0 or higher.

Lighting calculations of successive layers of mesh are calculated in separate calls of the GPU kernels in order to provide an appropriate sequence of calculations.

One of the obvious advantages of performing simulations on a graphics processor is that the results do not have to be transferred between computer RAM and graphics memory. CUDA can perform operations directly on the available texture memory.

5.5. Modeling light in the atmosphere

The realistic visualization of clouds needs to consider the phenomenon of light being scattered in the atmosphere, which affects the appearance of the sky and clouds.

Scattering, its intensity, and direction all depend on the wavelength and size of particles in the atmosphere. In general, this phenomenon is described by the Mie solutions of the Maxwell equations. In this simplified model, we use two popular approaches: approximation of Mie theory for small particles (called Rayleigh scattering) and Mie scattering for large particles.

Rayleigh scattering approximation is suitable for particles much smaller than the light wavelength, such as oxygen and nitrogen molecules found in the atmosphere. For a single particle of diameter d amount of light that is scattered at angle θ by a light beam with I_0 intensity for a distance from molecule R and the λ wavelength is equal to:

$$I = I_0 \frac{1 + \cos^2\theta}{2R^2} \left(\frac{2\pi}{\lambda}\right)^4 \left(\frac{n^2 - 1}{n^2 + 2}\right)^2 \left(\frac{d}{2}\right)^6,$$

where n is the refraction coefficient.

As we can see, λ occurs in the fourth power, which means that the amount of light that is scattered strongly depends on the wavelength. Shorter waves are more scattered, creating a visible blue color in the sky. Rayleigh scattering is also responsible for the yellow color of the Sun as well as its red color when it is low over the horizon (even though sunlight appears to be white in space).

The second phenomenon of the interaction of light with the atmosphere is Mie scattering for particles much larger than the wavelength, such as dust particles or water. In this case, the amount of scattered light weakly depends on the wavelength and is scattered mainly in the forward direction. This phenomenon is responsible for the brighter white sky near the sun as well as for white or gray clouds.

6. Rendering

The entire rendered scene consists of the three following elements: clouds rendered using a ray-casting algorithm, the surface of the Earth and the Sky.

The final image is generated in a series of successive steps:

1. Background rendering; calculation of the sky and Sun colors.
2. Terrain rendering.
3. Merging background images with the terrain; the result contains all of the elements of the final scene except for the clouds. At this stage, transparency effects are added to the atmosphere.
4. Clouds rendering clouds; the algorithm utilizes the data (esp. distances between elements) stored in the previous step.
5. Creating the final image – merging all of the images and the transformation from HDR color space to standard RGBA.

The dependencies between the steps are shown in Figure 2.

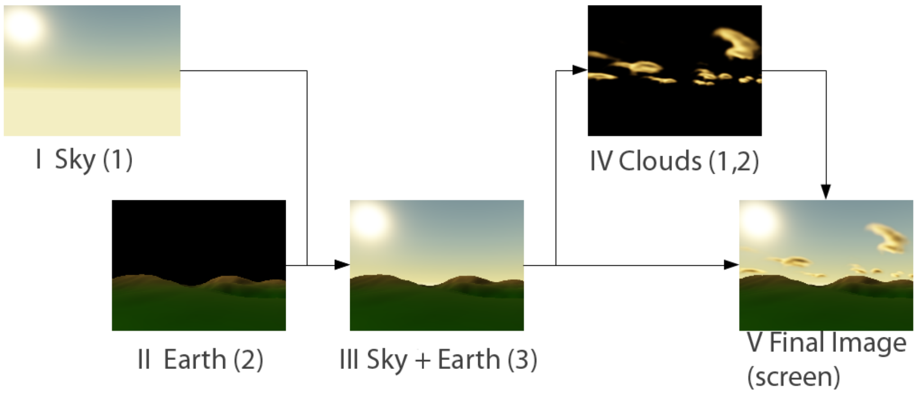

Figure 2. Stages of rendering a scene. Roman numbers indicate the image-generation step, while the numbers in parentheses show the number of the buffer in which the results are stored (three RGBA buffers are used).

6.1. Sky rendering

Rendering the sky uses an approach developed by O'Neil [10] and is based on the Nishita work [9] that simplifies the phenomena described in the previous section. This simplification is designed to quickly be evaluated in real time, giving us results that are close to reality (see Fig. 3).

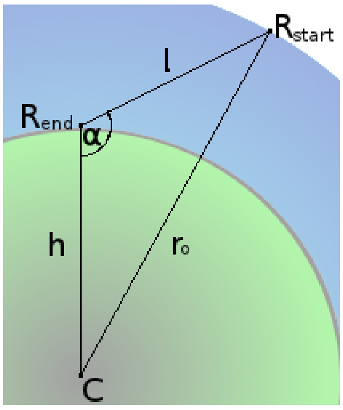

Figure 3. Sky rendering using a ray intersecting the atmosphere.

The color of the sky is calculated by summing up the number of samples along radius $\vec{R}$ whose beginning $\vec{R}_{start}$ lies on the edge of the atmosphere in the direction of the apex processed, and the end of $\vec{R}_{end}$ is equal to the position of the observer.

The radius of the Earth is denoted by r_i, the distance from the center of the Earth to the edge of the atmosphere by r_o, and the distance of the observer from the center of the Earth by h. The end point of the radius is equal to $\vec{R_{end}} = (0, 0, h)$.

The distance from the eye at R_{end} to the edge of the atmosphere at R_{start} can be easily calculated from the cosine theorem for the triangle, assuming that we know h, r_0 and $\alpha = \gamma + 90°$, where γ is the angle at which the observer looks over the horizon.

For each of the samples along l, we calculate the amount of light scattered towards the camera. The light value depends on the density of the atmosphere at this height, the angle between the direction of incidence of the light, and the radius. Each color component is calculated separately according to the amount of light scattered to the camera. The sum of the individual points of light gives the final color of the sky for the calculated point.

The resulting color obtained due to Rayleigh scattering and the angle between the radius and incoming light direction are copied to a fragment shader. On this basis, the Mie scattering is calculated. The final color of the light is the sum of these two types of scattering. The results are shown in Figure 4.

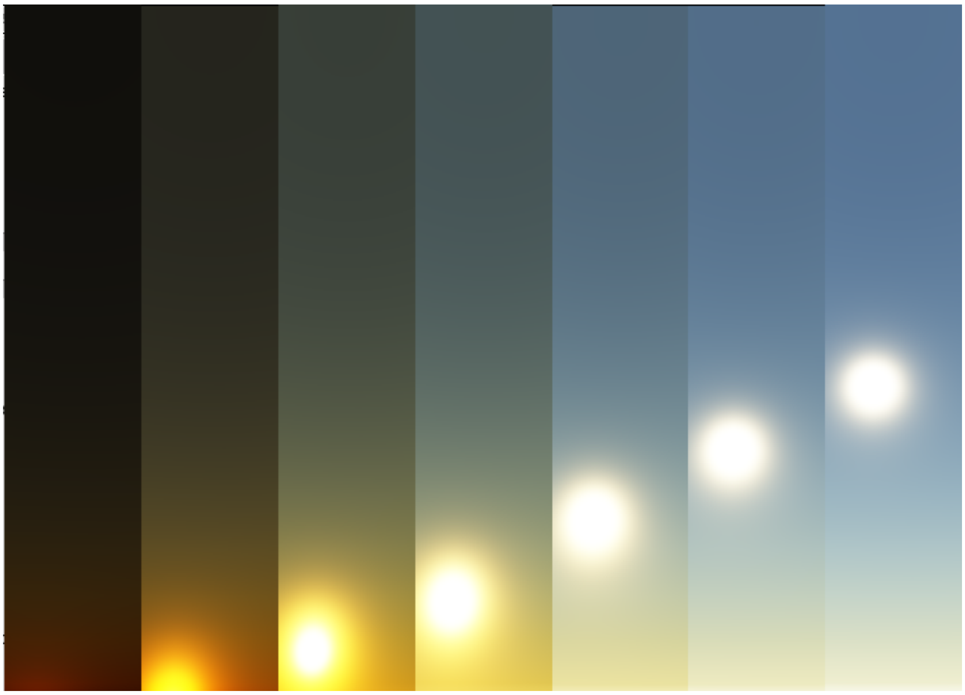

Figure 4. Images of the sky from horizon to zenith for different levels of the Sun above the horizon. From the left height of the Sun: $-5°$, $0°$, $5°$, $10°$, $20°$, $30°$ and $40°$.

6.2. Lighting conditions

Lighting affects the color of the rendered elements – clouds and terrain consist of ambient light that uniformly illuminate all of the elements and the scattered light coming directly from the light source that reaches the camera. Its amount depends on:

- the angle of incidence and shadowing by the clouds (for the terrain),
- occlusion by other parts of the clouds (for the clouds).

The colors and light intensity are calculated automatically by the same method as the color of the sky. This allows us to keep the scene illumination consistent, especially enabling us to show smooth illumination changes during sunrise or sunset.

Ambient light is obtained here as an average calculated in several equally spaced points in the sky. In these calculations, we do not include Mie scattering, because the Sun moving through one of the points taken into account would result in large changes in the color and intensity of light in a short amount of time. Instead, the color of the scattered light is calculated on the basis of a single point that lies in the direction of the light; in other words, where the sun is visible.

6.3. Cloud rendering

Usually, real-time computer graphics rendering is based on surfaces made of triangle meshes. This works very well in the most-common cases representing solids, but it is less suitable for partially transparent objects with fuzzy contours (such as clouds). There is no simple way to translate the results obtained from the simulation on the set of triangles to obtain smooth transitions between the values of the successive frames of the simulation.

For this reason, the cloud-rendering algorithm uses a ray-casting algorithm that is common in rendering volumetric data. This allows us to obtain very high image quality based on the data stored in the form of three-dimensional maps of density and color. Using of ray casting allows us to generate images seen from any direction as well as from any point in space (and also from inside the object).

6.3.1. Ray casting

The basic principle of the algorithm is to compute the color of the final portion of the image based on the values obtained from the three-dimensional data by sampling the data at a specified distance along the beam coming out of the camera and transmitted by rendering the image (Fig. 5).

One disadvantage of ray casting is the large quantity of operations that must be performed for each rendered fragment. This forces the use of a number of optimizations, which complicates the basic version of the algorithm.

6.3.2. Implementation

The algorithm is performed entirely on a GPU as a fragment shader used to determine the color fragments making up the rectangular area that limits a simulation box.

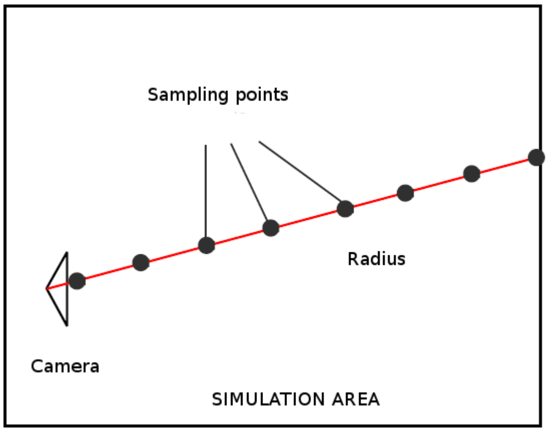

Figure 5. Ray casting – selected sample radius extending from the camera to the border of the simulation area (red line) with sampling points marked.

Vertices of this box are assigned to colors that match their coordinates. These colors are interpolated on the faces of the cuboid, which allows us to specify the coordinates of a point where the light ray leaves the rendered area for each fragment. The starting point of the ray is the camera position.

The input data are supplied in the form of a three-dimensional texture in which each texel contains four components of floating-point numbers. They contain the number of water droplets in each mesh node and the amount of light that reaches the node during the two most-recently-calculated simulation frames: F_i and F_{i-1}. The current value at time t is calculated according to the time assigned to each frame. Once the current time exceeds the value assigned to frame F_i, the values of frame F_{i-1} are swapped with the previously prepared values of frame F_{i+1}, and at the same time, the demand to calculate frame F_{i+2} is sent to the simulator.

Designated points on the ray are sampled, starting from those located closest to the camera. For each point $P_{1\dots n}$, the transparency t_I and color c_i of the corresponding section are calculated. The new values of resulting color C_i and transparency T_i are calculated using the following formulas:

$$C_i = C_{i-1} + T_{i-1}t_ic_i$$

$$T_i = T_{i-1}t_i$$

The initial values are 1 for transparency and 0 for color. With this method at the end of the loop, we get the color from range $[0, T_n]$. Final color C_k is obtained by linear scaling to the $[0, 1]$ range:

$$C_k = \frac{C_n}{T_n}$$

6.3.3. Ray casting optimizations

The implemented algorithm includes a number of optimizations that speed up its operations and improve the quality of the results.

Handling intersections with other objects

To properly handle the mutual occlusion of clouds and other objects, any samples that are hidden behind another element should be rejected. For this reason, the clouds must be rendered as the last element of the scene.

During the rendering of other objects, information about the distance from the camera is recorded for each fragment. As the scene does not contain any transparent objects other than clouds, it is sufficient to use the texture alpha channel to which the rendering is performed to store this information.

If the distance to obstacle d_0 is smaller than the length of the $\vec{V}$ ray, the end of the ray is moved so as to be at the point of intersection with the obstacle:

$$\vec{V}^* = \frac{\vec{V} d_0}{|\vec{V}|}$$

Limiting the number of samples

In some cases, if the camera is close to the edge of the rendered area or to the obstacles, the sampling points may be very close to each other. This causes unnecessary calculations more than a few samples in the space between the two adjacent nodes do not visibly improve the final result.

So, one can speed up the algorithm without any visible quality loss as a result of introducing a minimum distance between points. For short-rays where the distance is smaller than this value, the number of samples is reduced according to $n^* = \frac{n}{|\vec{V}|}$

Early ray termination

In situations where the scene contains dense clouds, the space situated behind them can be completely obscured. In such cases, sampling invisible space does not affect the final result and should be terminated.

For this purpose, the ray-casting loop is aborted if transparency falls below a T_{min} value, usually slightly greater than 0. Because subsequent transparency values (T_i) are obtained by multiplying the previous value by a number from the $(0, 1]$ range, it is possible that $T_{min} = 0$ and the threshold can never be reached.

Using too high T_{min} is also not recommended because it sharpens the cloud edges, which become completely opaque in places where the background should still be seen. In practice, with a value of 0.01, early ray termination is not noticeable, while the speed of the algorithm is increased.

Since the value of $T_{min} > 0$, this could create a situation in which the background is always seen slightly through the clouds, regardless of their color and thickness.

To avoid this effect, transparency range $[T_{min}, 1]$ is mapped to $[0, 1]$, and final opacity T_k is equal to:

$$T_k = \frac{T_i - T_{min}}{1 - T_{min}}$$

Figure 6 shows that this technique reduces the sampling area obscured by a cloud, allowing for the significant reduction of the total number of operations performed for the scene with many objects near the camera.

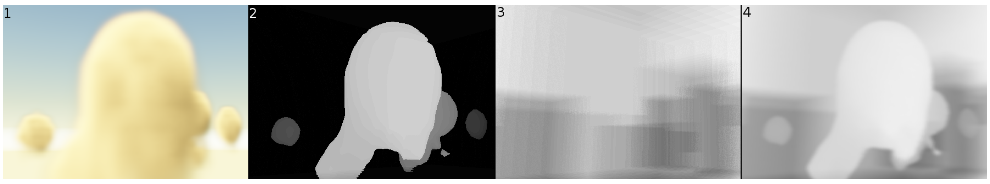

Figure 6. Ray-casting optimizations allow for the omission of unnecessary sampling points. The brightness of the pixels in the image corresponds to the number of collected data samples: the darker the gray shade, the more samples that are applied. 1) standard view; 2) early ray termination; 3) occupancy map (boundaries of cubic map elements are visible); 4) early ray termination and occupancy map.

Occupancy map

The use of occupancy maps allows for the efficient rendering of data that contains a large amount of empty space by enabling the abiity to quickly jump through the empty areas without the time-consuming re-sampling and interpolation from the texture data. The mesh has been divided into cubes with an edge length of 4. For each, if any mesh node belonging to the cube has a non-zero number of water droplets, it is flagged as occupied; otherwise, it is labeled as free. In practice, instead of comparing to zero, we use a comparison to a very small positive value low enough that the omission of such an element is not noticeable.

The map is created each time after the calculation of a frame is done. It is the logical sum of two successive frames used for rendering. It is passed to the shader as a one-component three-dimensional texture for further rendering. In each iteration of the main ray-casting loop, the corresponding cell in an occupancy map is checked before loading the values from the texture data. If the cell has a state of 0, the loop is skipped, and the algorithm proceeds to the next step.

Regardless of the value read from the map, the new reading is performed only if the processed point falls into a different cell than the previous one. This allows for a further reduction of the number of reads from the texture.

In his solution, Meissner [5] proposes a hierarchical division of the mesh into unit cubes grouped into larger ones with a side length equal to 4. In this approach, collecting data from a small cube is done using fast bit operations. However, for the application described herein, such a method proved to be less efficient than a simple single-level grid division. The tested cube sizes (2, 4, and 8) showed that the the highest efficiency was achieved for the edges equal to 4.

As can be seen in Figure 6, the occupancy map is very effective in situations where the rendered scene contains a lot of empty space. This optimization complements the early ray termination, which gives the best results when the scene is densely filled with clouds.

Interleaved sampling

The interleaved sampling mechanism first proposed for a GPU in [13] allows us to avoid the artifacts described above. The idea is that a sampling vector moves at a pace depending on the position on the screen of the rendered part of the image, therefore the sampling process may be perceived as less regular.

The original method has been dedicated to work with opaque objects and assumes the calculation of vector movements according to a small regular pattern. When applied to transparent clouds in which the final color is made up of many samples, the use of a regular pattern gives new artifacts. Better results can be achieved using the random offset in a predetermined range. The best results are obtained when the range of the random offset is equal to the spacing between successive samples, which results in a uniform distribution of the samples in space.

Applying the interleaved sampling to a small number of samples can lead to noise arising from the fact that the color of the adjacent portions is calculated on the basis of different locations in the input data. For clouds, to reduce the visual effects of the noise, one can use a Gaussian filter to blur the image.

Figure 7 shows the results with the interleaving mechanism switched on and off. In a situation in which the neighboring sampling points are close to each other and the space between the two mesh nodes is sampled several times, the interleaved sampling application does not affect the results. However, in a situation with small number of samples, the difference becomes very apparent. An image obtained by using the described technique is substantially better than that which is achieved without it.

This technique can, therefore, be successfully used to improve the performance in situations where the performance of the hardware is too low to perform a rendering of a sufficiently large number of samples.

6.4. Terrain rendering

The surface of the Earth is an example of how to add new elements rendered in the standard way to the algorithm used to render clouds (Fig. 8). Using this method in the scene, one can add any other elements (including trees, buildings, or characters).

The terrain is rendered as a rectangular grid of triangles whose vertical coordinate is calculated based on the height maps loaded from an image file. The color depends on the height of the terrain and is calculated on the basis of a one-dimensional texture.

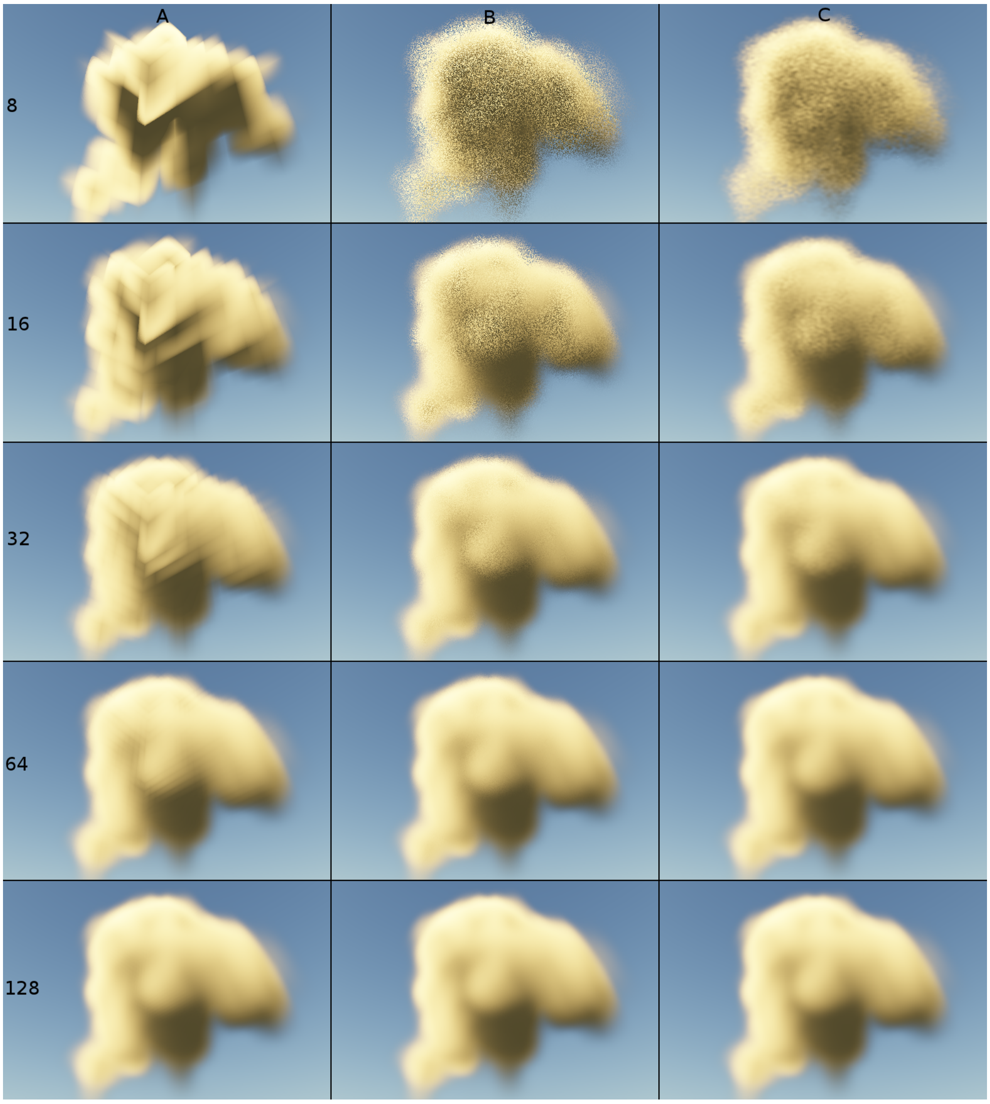

Figure 7. Effect of interleaving mechanism for rendering results for different sampling density. The numbers in the rows indicate the number of samples. Column description: A) interleaving switched off; B) interleaving switched on; C) interleaving and blur switched on. The images obtained from the data of size 64 × 64 × 32.

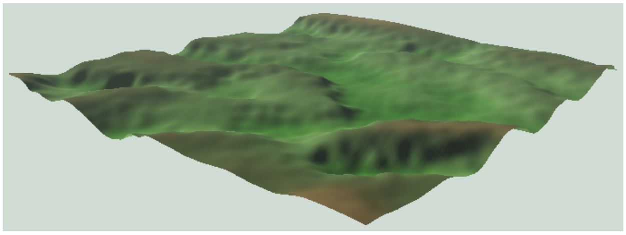

Figure 8. Render of the terrain – a sample view.

The effect of shadows cast on the terrain through the clouds is obtained by calculating the coordinates of a point $P = (p_x, p_y, p_z)$ on the terrain. According to a given light vector $vecL$, we calculate the shadow point in the lowest data layer.

$$\vec{P}_{shadow} = \vec{P} + (p_z \vec{L})$$

The amount of light reaching the earth is calculated on the basis of the value taken from the texture data in the same way as in the cloud rendering.

6.5. Final image generation

The final image is obtained by superimposing the images of the sky, terrain, and clouds with regard to the transparent parts of these images. Subsequently, the color is converted from the HDR range to a standard $[0, 1]$ interval according to the following formula:

$$C_k = e^{-EC_{hdr}}$$

where E is the camera-sensitivity coefficient.

7. Performance and sample results

The following tables 2, 3, 4 present the data on the speed of the simulation and rendering for several different scenarios.

Table 2
The dependence of speed of simulation vs number of grid points.

CPU time (ms)	GPU time (ms)	Grid size (number of points)
12	3	$16 \times 16 \times 16(2^{12})$
33	6	$32 \times 32 \times 16(2^{14})$
213	29	$64 \times 64 \times 32(2^{17})$
828	109	$128 \times 128 \times 32(2^{19})$
3112	425	$256 \times 256 \times 32(2^{21})$
5982	881	$512 \times 512 \times 16(2^{22})$

Table 3

Rendering speed versus image resolution and number of samples. Rendering uses data of $64 \times 64 \times 32$ (2^{17} points).

Rendering time (ms)	Number of samples	Resulting image resolution
22	32	800×600
37	64	800×600
65	128	800×600
116	256	800×600
5	32	400×300
9	64	400×300
17	128	400×300
30	256	400×300

Table 4

Simultaneous rendering and simulation.

Only render	CPU simulation		GPU simulation		Resolution image	Number samples	Mesh size
	simul.	render	simul.	render			
31	252	273	35	43	800×600	64	$64 \times 64 \times 32$
45	940	609	52	69	800×600	128	$128 \times 128 \times 32$
65	4217	1547	68	139	800×600	256	$256 \times 256 \times 32$

The tests were performed on a computer with a dual-core 2.2GHz AMD Athlon processor and a rather-modest graphics card (an Nvidia GeForce GT 430 containing 2 multiprocessors with 48 cores each).

A description of the experiments is as follows:

1. Simulation speed dependent on the number of grid points in the mesh (Table 2). With rendering switched off, the simulation is performed during 100 steps with the same parameters for different mesh sizes.
2. Rendering speed dependent on the resolution of the image and number of sampling points (Table 3). With disabled simulation, rendering the same previously generated data is performed for different sizes of the resulting image and the size parameter for different numbers of the collected data samples.
3. Simultaneous rendering and simulations (Table 4). The speed of rendering and simulation for a single frame is performed for three cases: only rendering enabled, integrated rendering, and simulation on a CPU and GPU.

Conclusions:

- As expected, the simulation scales linearly depending on the number of grid points for both simulations performed on the CPU or the GPU.
- Cloud-rendering algorithm scales linearly with the number of rendered points and number of data samples collected for each point.
- On the hardware used for tests, the simulation runs much faster on the graphics card than on the main processor.

- The simultaneous execution of simulation and rendering has a much greater impact on the time required to execute individual components when performing simulations on the GPU. For the CPU, the simulation load is distributed between the main processor and the graphics processor, and the overhead is much smaller.
- For the smooth animation of cloud evolution, it is necessary to calculate the simulation frame every few seconds (and tens of frames per second for the rendering part). At the same time, the number of data samples for the rendering algorithm should not be less than the largest dimension of the mesh. The largest network for which we could perform calculations on our hardware fast enough was a size of$256 \times 256 \times 32$. Such a mesh is sufficient to obtain good-quality results.

A few examples of the results obtained using the implemented application are presented in Figures 9 through 14.

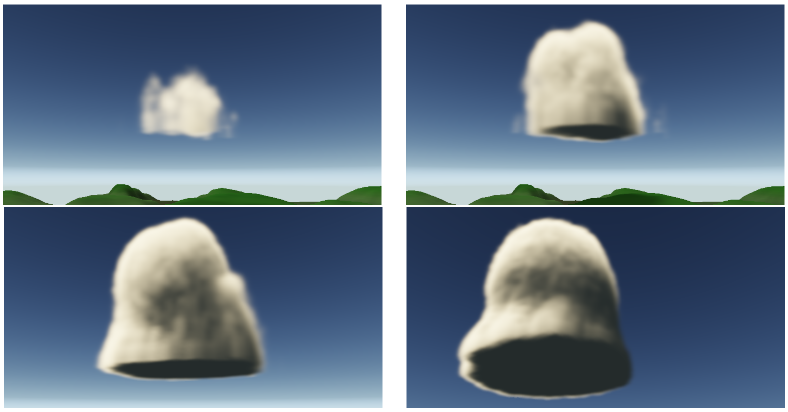

Figure 9. The evolution of clouds – in 120, 150, 200, and 300 steps of simulation (relative time units).

Figure 10. Cirrocumulus-type clouds.

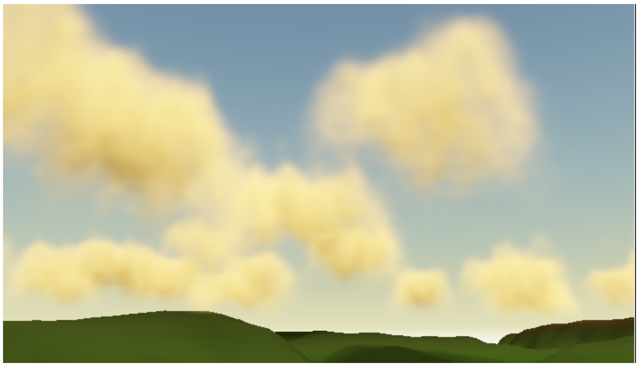

Figure 11. Light cumulus clouds.

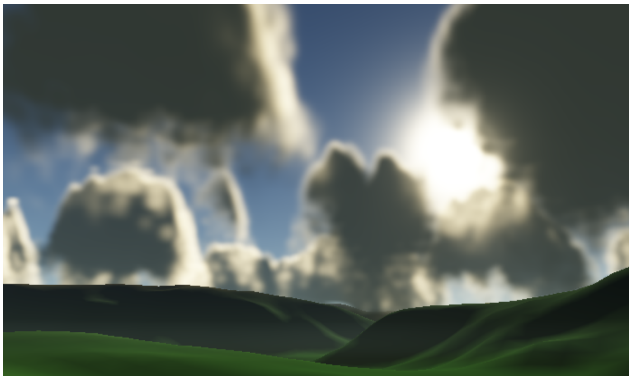

Figure 12. More-developed and denser cumulus clouds.

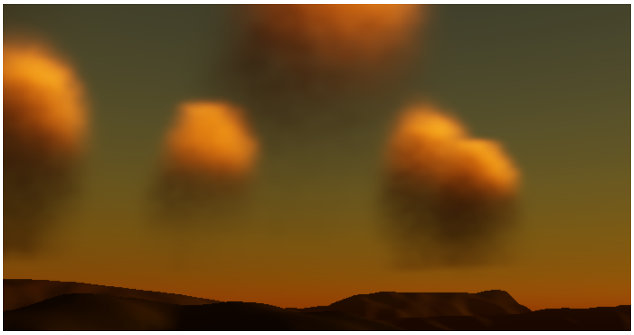

Figure 13. Clouds illuminated during sunset.

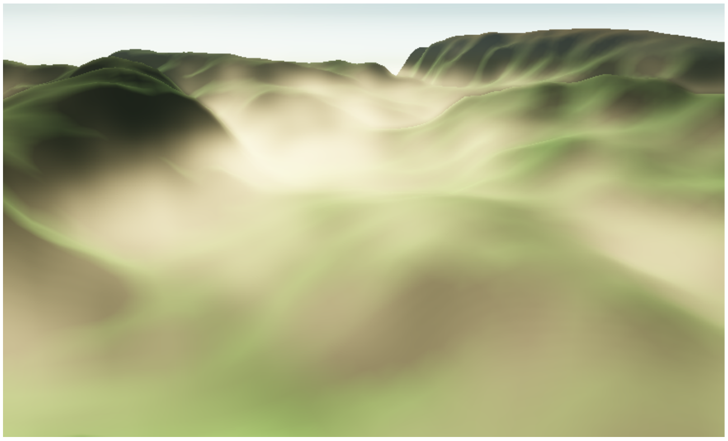

Figure 14. With the proper selection of parameters (high humidity combined with a low temperature near the ground), simulation can be used to generate heterogeneous fog. The illustration shown a residual mist in a valley.

8. Conclusion

The results obtained show that it is possible to obtain a smooth animation of cloud evolution in real time by using atmosphere simulation together with a cloud-rendering algorithm. The advantage of this method is that it is fully three-dimensional, which is important for applications where the camera is not connected to the ground and can pass through the layers of clouds.

The proper selection of the simulation parameters allows for the generation of different types of clouds. A large number of these parameters makes their selection difficult.

The method applied is demanding when it comes to the amount of memory consumption and CPU time. However, given the rapid development of computer capabilities and ease of adjustment of both simulation and rendering systems as well as the available resources at the expense of the quality of the results, it seems that the method may be useful where it is necessary to generate a dynamically changing sky.

The resulting cloud images are of good quality but still lack details as compared to actual clouds. Obtaining an image similar to the real one requires a greater computational mesh as well as a more-advanced atmosphere model.

Acknowledgements

The research presented in this paper was partially supported by the Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.015 (statutory project).

References

[1] Bouthors A., Neyret F.: Modeling Clouds Shape. In: Alexa M., Galin E. (eds.), *25th European Conference Eurographics 2004 (Short papers)*. Eurographics, 2004.

[2] Dobashi Y., Kaneda K., Yamashita H., Okita T., Nishita T.: A simple, efficient method for realistic animation of clouds. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp. 19–28, 2000. `http://dx.doi.org/10.1145/344779.344795`.

[3] Harris M.J., Baxter W.V., Scheuermann T., Lastra A.: Simulation of cloud dynamics on graphics hardware. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 92–101, 2003. `http://dl.acm.org/citation.cfm?id=844174.844189`.

[4] Kaneko K.: Simulating Physics with Coupled Map Lattices-Pattern Dynamics, Information Flow, and Thermodynamics of Spatiotemporal Chaos. In: Kawasaki K., Onuki A., Suzuki M. (eds.), *Pattern Dynamics, Information Flow, and Thermodynamics of Spatiotemporal chaos*, pp. 1–52. World Scientific, Singapore, 1990.

[5] Meißner M., Doggett M.C., Hirche J., Kanus U., Straßer W.: Efficient Space Leaping for Raycasting Architectures. In: Mueller K., Kaufman A.E. (eds.), *Proceedings of the Joint IEEE TCVG and Eurographics Workshop on Volume Graphics in Stony Brook, New York, USA, June 21–22, 2001*. Eurographics Association, 2001. `http://dx.doi.org/http://www.eg.org/EG/DL/WS/VG01/Meissner/paper_full.pdf`.

[6] Miyazaki R., Yoshida S., Dobashi Y., Nishita T.: A Method for Modeling Clouds based on Atmospheric Fluid Dynamics. In: *Proceedings Pacific Graphics 2001*, pp. 363–372, 2001.

[7] Nagel K., Raschke E.: Self-organizing criticality in cloud formation? *Physica A: Statistical Mechanics and its Applications*, vol. 182(4), pp. 519–531, 1992. `http://dx.doi.org/10.1016/0378-4371(92)90018-L`.

[8] Neyret F.: Qualitative Simulation of Cloud Formation and Evolution. In: Thalmann D., de Panne M.V. (eds.), *8th Eurographics Workshop on Computer Animation and Simulation (EGCAS'97)*, Eurographics, Springer Wein, New York City, NY, pp. 113–124, 1997.

[9] Nishita T., Shirai T., Tadamura K., Nakamae E.: Display of The Earth Taking into account Atmospheric Scattering. In: *Proceedings of SIGGRAPH'93*, pp. 175–182, 1993.

[10] O'Neil S.: *GPU Gems 2*, chap. 16. Accurate atmospheric scattering, Addison-Wesley Professional, 2005.

[11] Perlin K.: An Image Synthesizer. In: *Proceedings of SIGGRAPH'85*, pp. 287–296, 1985.

[12] Perlin K.: Improving Noise. In: *Proceedings of SIGGRAPH'02*, pp. 681–682, 2002.

[13] Scharsach H.: Advanced GPU Raycasting. In: *Proceedings of CESCG 2005*, pp. 69–76, 2005.

[14] Stam J., Fiume E.: Turbulent wind fields for gaseous phenomena. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pp. 369–376. ACM, New York, 1993. `http://dx.doi.org/10.1145/166117.166163`.

[15] University of Illinois WW2010 Project. `http://ww2010.atmos.uiuc.edu/\%28Gh\%29/guides/mtr/cld/cldtyp/home.rxml`.

Affiliations

Paweł Kobak
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, pawelkobak@gmail.com

Witold Alda
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, alda@agh.edu.pl

Received: 9.03.2015
Revised: 8.05.2017
Accepted: 9.05.2017