# Efficient Algorithms for Real-Time GPU Volumetric Cloud Rendering with Enhanced Geometry

**Article** *in* Symmetry · April 2018

DOI: 10.3390/sym10040125



**2 authors**, including:

Carlos Jiménez de Parga
National Distance Education University
**5** PUBLICATIONS **4** CITATIONS

SEE PROFILE

symmetry

MDPI

Article

# Efficient Algorithms for Real-Time GPU Volumetric Cloud Rendering with Enhanced Geometry

**Carlos Jiménez de Parga * and Sebastián Rubén Gómez Palomo**

Software Engineering and Computer Systems Department, National Distance Education University (UNED), 28040 Madrid, Spain; sgomez@issi.uned.es

* Correspondence: carjimenezdeparga@cartagena.uned.es

Received: 6 March 2018; Accepted: 18 April 2018; Published: 20 April 2018


**Abstract:** This paper presents several new techniques for volumetric cloud rendering using efficient algorithms and data structures based on ray-tracing methods for cumulus generation, achieving an optimum balance between realism and performance. These techniques target applications such as flight simulations, computer games, and educational software, even with conventional graphics hardware. The contours of clouds are defined by implicit mathematical expressions or triangulated structures inside which volumetric rendering is performed. Novel techniques are used to reproduce the asymmetrical nature of clouds and the effects of light-scattering, with low computing costs. The work includes a new method to create randomized fractal clouds using a recursive grammar. The graphical results are comparable to those produced by state-of-the-art, hyper-realistic algorithms. These methods provide real-time performance, and are superior to particle-based systems. These outcomes suggest that our methods offer a good balance between realism and performance, and are suitable for use in the standard graphics industry.

**Keywords:** ray-tracing; algorithms; fractals; volumetric rendering; radiometry; computational geometry

## 1. Introduction

Dynamic rendering of realistic clouds 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 photo-realistic radiometric functions for lighting effects, which can prevent real-time rendering in modern graphics processing units (GPUs) [1,2], causing a lack of realism [3] or the absence of animation effects [4]. As an alternative approach, we propose efficient methods for atmospheric cloud geometry generation using pseudo-spheroids and fractals, which produce the stochastic and irregular shapes of natural phenomena, as a baseline for an easy-to-control framework for primitive cloud animation. In addition, our methods allow for approaching, turning around, and passing through gaseous shapes by use of the volumetric rendering technique. The aim of our investigation is to confirm the hypothesis that it is possible to create an efficient, lightweight, and user-friendly model of real-time cloud rendering with an optimum balance between realism and performance, to be applied in the entertainment and educational industries even in situations where the graphical hardware is not very powerful. For this reason, our hypothesis is based on a multi-core processing approach that has arisen in recent years and the possibility of optimizing the complexity of the referenced algorithms with GPU and central processing unit (CPU) programming techniques.

Our methods are performed using the ray-tracing technique, based on the work of Appel [5] and Whitted [6], and the emulation of the density of water vapour inside a cloud by applying our base algorithm to generate a gaseous body inside implicit surfaces. To maximize the performance of the pre-computing phase, an original algorithm is used to prevent duplicate ray-marching inside

overlapping volumes, which we have called *no duplicate tracing* (*NDT*). This achieves fewer loop iterations by discarding the overlapping parts of spheres and reduces computing times by half. We have also developed new approaches for lighting and shading that were inspired by the paradigms of distinguished authors, with optimizations to reduce the cost of light-scattering calculations. The use of boundary representations based on Smits' algorithm [7] for the ray-marching volume, along with Euclidean equations for ray–spheroid collision, allows for minimal linear complexity when searching for primitives, hence avoiding the use of hard-to-code space partitioning algorithms. An improved cumulus generation algorithm based on Gaussian distributions simulates the stochastic nature of clouds. Additionally, an innovative application of L-system grammar rules, with random parameters to emulate the fractal nature of clouds, is compared to the method used by Kang et al. [8]. We also provide an innovative contribution to cloud rendering based on triangulated three-dimensional meshes, obtained from standard editors, to produce clouds in the shapes of recognizable entities. Complexity analysis and benchmarking of the proposed algorithms demonstrate that there are more benefits than disadvantages in comparison to other particle and slow, hyper-realistic implementations. Hence, our algorithms are suitable for application in the standard industry.

## 2. Related Works and Main Contributions

Researchers studying computer-generated clouds have developed different approaches to model the natural phenomena. Huang et al. [1] reports on the two main groups of cloud rendering methods: *Physics-Based Models* and *Ontogenetic Models*. In the first approach, atmospheric equations of cloud advection, thermodynamics, radiometry, and fluid dynamics are applied [3,9]. In the second method, a formal geometric visualization of the cloud in an artistic simulation, using minimal physical characteristics, is employed. The following paragraphs describe the methods used to realize natural-looking clouds along with our contributions to each of them:

- *Texturized primitives:* In this option, basic surfaces like skydomes or skyboxes are texturized with an omni-directional true color capture of the sky. Other implementations use procedural textures applied to spheres and ellipsoids, which was the starting point for cloud generation in the early years with the works of Gardner [10] and Max [11]. These methods may be considered as obsolete with modern GPUs, however, they are employed in smaller 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 [4]. The main limitation of these methods is the inability to approach, turn around, or traverse gaseous bodies, and additionally the lack of animation of the different cloud parts. We decided not to use this method due to the aforementioned limitations.
- *Particle systems:* Since 2002, researchers have tried to achieve simulations of clouds using basic quads or triangle surfaces with Gaussian textures [1]. Harris conceived a new algorithm using particles with an efficient multiple scattering illumination system, which was based on Max equations [12]. The resulting rendering was improved with the use of impostors [3,13,14]. 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. We decided to use pseudo-spheroids as the basic primitive instead of particles to allow for animation of the cloud with reduced system overhead. The realism of each pseudo-spheroid is better than using a collection of texturized particles. This method is very convenient to produce time-evolving cumuliform clouds.
- *Geometry distortion:* The basic idea behind this technique consists of drawing a complex cloud mesh with a group of megaparticles as explained by Ebert [15] and Engel [16]. Then, a lighting algorithm, like *Phong* or *Gouraud* shading, is used on the geometry. After this step, the cloud map is blurred 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 two-channel fractal/noise texture to obtain a distortion offset. Afterwards, the blurred cloud map is sampled using these

texture coordinates and the distance from the quad to the camera. Optionally, a radial blur may be performed to soften the resulting image. In the final stage, the render target is merged to the back buffer. This method is midway between the particle systems and volumetric rendering techniques, with good performance and easy shading. However, it lacks accurate realism and is not suitable for cloud shapes other than cumulus. We have decided not to use this method because it is difficult to adapt for arbitrary-shaped clouds.

- *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 primary previous studies on volumetric rendering are found in References [17,18]. The methods proposed in this paper are based on volumetric rendering. This technique is the same as that used to obtain two-dimensional (2D) medical images from 3D magnetic resonance (MRI) or computed tomography (CT) data. Volumetric rendering of clouds still requires 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 [19], Elek et al. [20], Dobashi et al. [21], and Goswami-Neyret [22], while further contributions include the latest cutting-edge findings in light scattering made by Klehm et al. [23] and Peters et al. [24], with very high performance at 1920 × 1080 pixels. We contributed to the state-of-the-art by improving the performance of existing algorithms and developing new efficient algorithms and methods to reduce the physics workload of many slow hyper-realistic implementations, without loss of quality. These methods are well suited for animation and real-time rendering on standard computers. Table 1 summarizes the main features and shortcomings of the mentioned methods.

**Table 1.** Pros and cons of each method.

| | Texturized Primitives | Particle Systems | Geometry Distortion | Volumetric Rendering |
|---|---|---|---|---|
| **Procedural** | ✓ | ✗ | ✗ | ✓ |
| **Animation** | ✗ | ✓ | ✓ | ✓ |
| **Traverse** | ✗ | ✓ | ✗ | ✓ |
| **Interactive** | ✓ | ✓ | ✓ | ✓ |
| **Turn around** | ✗ | ✓ | ✓ | ✓ |
| **All kind of clouds** | ✓ | ✗ | ✗ | ✓ |
| **Realism** | ✓ | ✗ | ✓ | ✓ |
| **Performance** | ✓ | ✓ | ✓ | ✗ |
| **State-of-the-art** | ✓ | ✗ | ✗ | ✓ |

## 3. Materials and Methods

Volumetric rendering applications require extensive computing. Any small improvement in the algorithms produces a substantial overall increase in speed. The following are our contributions and improvements to minimize rendering efforts in a state-of-the-art trend.

### *3.1. 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 real-time cloud rendering. However, ray-tracing of three-dimensional points in real time requires hardware support by GPU shader technology that parallelizes the ray-marching of discrete frame buffer pixels using multiple processing elements.

To create an efficient vapour density simulation, a three-dimensional cube of 64 × 64 × 64 single-precision floats is filled with uniform random noise pre-calculated in the host before passing it to the GPU. This cube is used to generate fractal Brownian motion (fBm) noise, as shown in Figure 1.

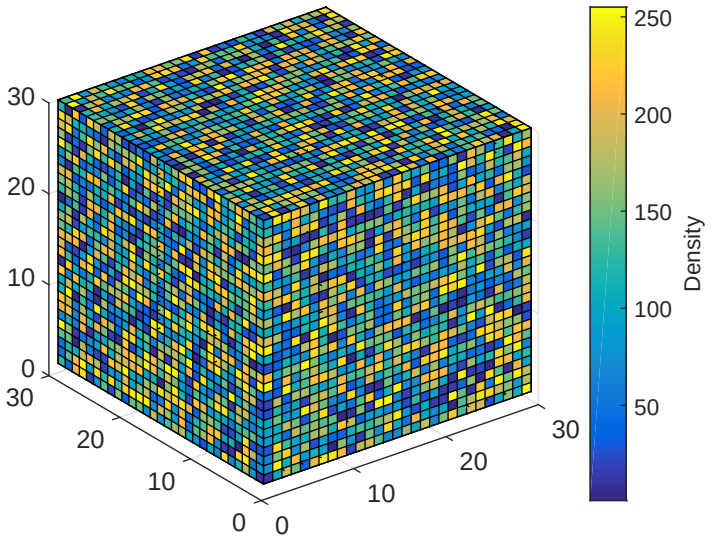



**Figure 1.** The uniform noise in the color scale plot shows the irregular density of water droplets in a cloud hypertexture. In the MATLAB four dimensional plot, it is possible to observe the nature of atmospheric vapour in the color values in the cube.

The uniform noise is transformed into a Gaussian-like noise using the fractal Brownian motion (fBm) method (Figures 2 and 3) described on Iñigo Quilez's homepage (http://www.iquilezles.org/www/articles/morenoise/morenoise.htm (6 April 2018)). Basically an fBm can be implemented as a fractal sum of *Perlin* noise functions [25]. The *Perlin* noise is a pseudo-random 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 a procedural texture in computer graphics to emulate controllable pseudo-random appearances, with the aim of generating a wide variety of object surfaces, for example fire, smoke, and clouds. This noise usually imitates textures in nature due to its stochastic properties. The implementation implies three steps: (1) definition of an n-dimensional grid with random gradient vectors, (2) computation of the dot product between the gradient vectors for distance calculations, and (3) interpolation of the mentioned dot products. The algorithm cost is $O(2^n)$ where $n$ is the number of dimensions (https://rosettacode.org/wiki/Perlin_noise (6 April 2018)).

Let $w$ be the octave scale factor, $s$ the noise sampling factor, and $i$ the octave index. The fBm equation is then defined as:

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

where we choose $w = 1/2$ and $s = 2$.

Each iteration is called an octave. In this case, our Algorithm 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 and decreases the overall frame performance. To improve performance, an unrolled version of the previous summation is implemented in our fragment shader code.

Other authors (see for example Reference [26]) also make use of fractional octave division in their research with respect to 2D flat animation of clouds.

The fBm noise is calculated for each unit of the frame buffer along the ray-tracing 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 the entire scene, a simple ray marching algorithm may be used, as explained on the next page.

**Algorithm 1** Basic Volumetric Cloud Rendering in GPU

**function** GETCANDIDATES($rayOrigin, \vec{rayDirection}, i$)
  **for** $j \leftarrow 0 <$ number of spheres in boundingbox[i] **do**
    $\vec{temp} \leftarrow rayOrigin - \vec{sphereCenter}$
    $a \leftarrow \vec{rayDirection} \cdot \vec{rayDirection}$ {Dot products}
    $b \leftarrow 2.0 \cdot \vec{rayDirection} \cdot \vec{temp}$
    $c \leftarrow \vec{temp} \cdot \vec{temp} - radius^2$
    $\Delta \leftarrow b^2 - 4ac$
    **if** $\Delta > 0$ **then** {There is a collision}
      $\sigma \leftarrow \sqrt{\Delta}$
      $\lambda_{in} \leftarrow \frac{-b-\sigma}{2.0 \times a}$
      $\lambda_{out} \leftarrow \frac{-b+\sigma}{2.0 \times a}$
      $candidates[k] \leftarrow (\lambda_{in}, \lambda_{out}, j, i)$
      $k = k + 1$
    **end if**
  **end for**
  **return** (candidates,k)
**end function**
**function** SORTCANDIDATES(candidates, n)
  **for** $k \leftarrow 1 < n$ **do** {Insertion-sort algorithm}
    aux = candidates[k]
    $h = k - 1$
    **while** $((h >= 0) and (aux_{\lambda_{in}} < candidates[h]_{\lambda_{in}}))$ **do**
      candidates[$h + 1$] = candidates[h]
      $h = h - 1$
    **end while**
    candidates[$h + 1$] = aux
  **end for**
**end function**
**function** RAYTRACE($rayOrigin, \vec{rayDirection}$)
  $B \leftarrow boundingboxDetection(rayOrigin, \vec{rayDirection})$
  $\tau \leftarrow 1$
  $R \leftarrow (0, 0, 0, 0)$ {Consider alpha-channel}
  **for each** (i in B) **do**
    $(C, n) \leftarrow getCandidates(rayOrigin, \vec{rayDirection}, i)$
    $candidates \leftarrow sortCandidates(C, n)$
    **for** $j \leftarrow 0 < n$ **do**
      $\lambda \leftarrow candidates[j]_{\lambda_{in}}$
      $\lambda_{out} \leftarrow candidates[j]_{\lambda_{out}}$
      **while** $\lambda \leq \lambda_{out}$ **do** {Trace pseudo-spheroid}
        $rayPos \leftarrow rayOrigin + \lambda \cdot \vec{rayDirection}$
        $\rho \leftarrow fbm(rayPos)$
        $\gamma \leftarrow e^{-\|rayPos - \vec{sphereCenter}\| / (r((1-\kappa) + 2k\rho))}$
        **if** $\rho < \gamma$ **then**
          $R \leftarrow R +$ lighting($\rho, rayPos, \vec{rayDir}, \vec{sunDir}$,
            $voxelGrid[candidates[j]_i], sunColor$)
          **if** $\tau < 10^{-6}$ **then**
            **break for** (36:)
          **end if**
        **end if** {Level of Detail (LOD)}
        $\lambda \leftarrow \lambda + A \cdot e^{-}(\|rayOrigin - \vec{cloudCenter}\| \cdot \delta)$
      **end while**
    **end for**
  **end foreach**
  **return** $R$ {Blend with the sky}
**end function**

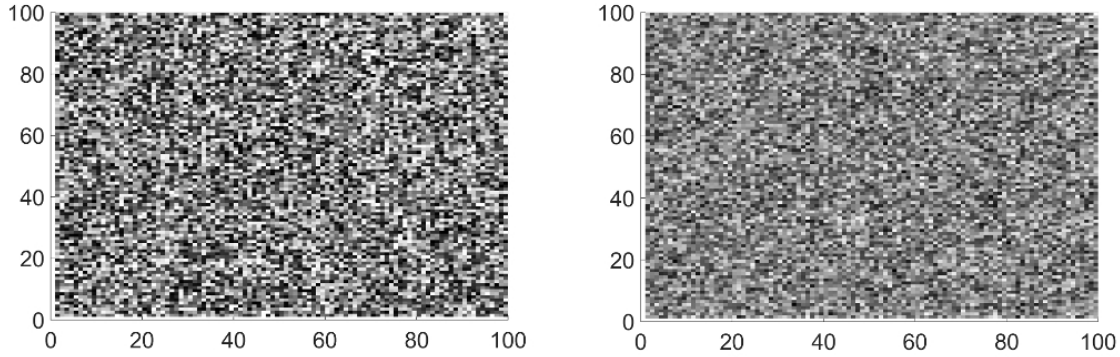



**Figure 2.** Bi-dimensional representation of uniform noise (**left**) and fractal Brownian motion (fBm) noise (**right**). The base noise on the left is sharper while the fBm is softer due to the octave accumulation.

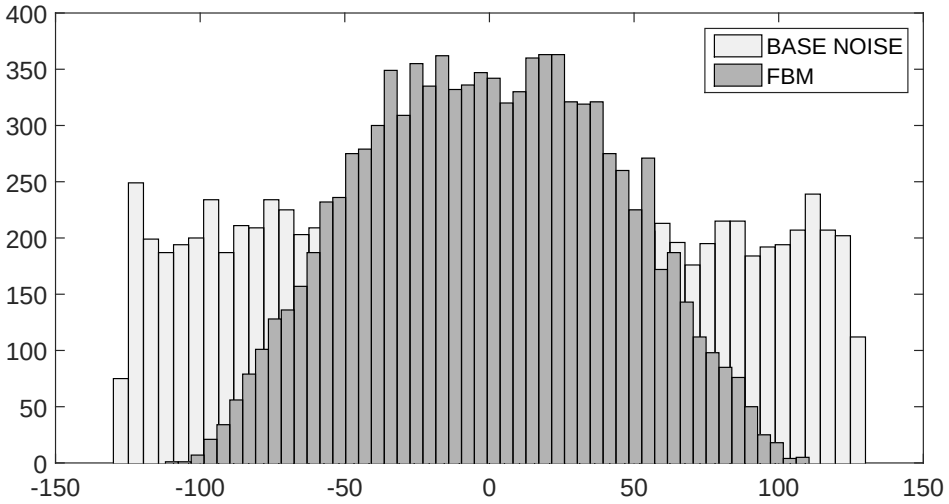



**Figure 3.** Histogram of uniform noise (light gray) and fBm noise (dark gray). The fBm noise has a Gaussian-like distribution.

### *3.2. Cloud Tracing*

Rendering a cloud involves tracing rays through a volume delimited by a set of implicit surfaces, while sampling the noise texture. The noise texture is only sampled when the ray traverses the volume defined by a set of spheres. The pseudocode scheme in Algorithm 1 is an improved version of Apodaca's code [27] and illustrates the execution over a list of N randomized bounded-surface volumes. The use of spheroids and ellipsoids instead of other surfaces is also cited by Gardner [10], and Elinas and Stürzlinger [28]. To discard positions outside the cloud and thus reduce the computational effort, we iterate over the pseudo-spheroids contained in the corresponding bounding box only. In addition, a short list of pseudo-spheroids that intersect the ray is created and sorted with the *insertion sort algorithm*. This collision method requires the computation of the discriminant of the quadratic equation, resulting from the equality of the implicit formulas for the rays and spheres, and uses reference code based on Shirley's book [29].

For every pixel in the frame buffer, the ray is traced from back to front to calculate the noise, with the condition that the density value is smaller than $\gamma$. This variable is modulated as a function of the position inside the pseudo-spheroid with a random component to produce the effect of the cloud surface as shown in Equation (2):

$$\gamma \leftarrow e^{\left(\frac{-\|rayPos-\vec{sphereCenter}\|}{radius\times\left(\left(1-\kappa\right)+2\kappa fbm(x,y,z)\right)}\right)}. \quad (2)$$

The previous equation filters the unit volume densities and generates a pseudo-spheroid by scaling the nominal sphere radius by ± 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. Additionally, an improved level of detail (LOD) equation controls the increment on the Euclidean straight line [30] 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 balances the higher number of pixels receiving a trace from inside the volume with the lower requirements for detail when traversing the cloud.

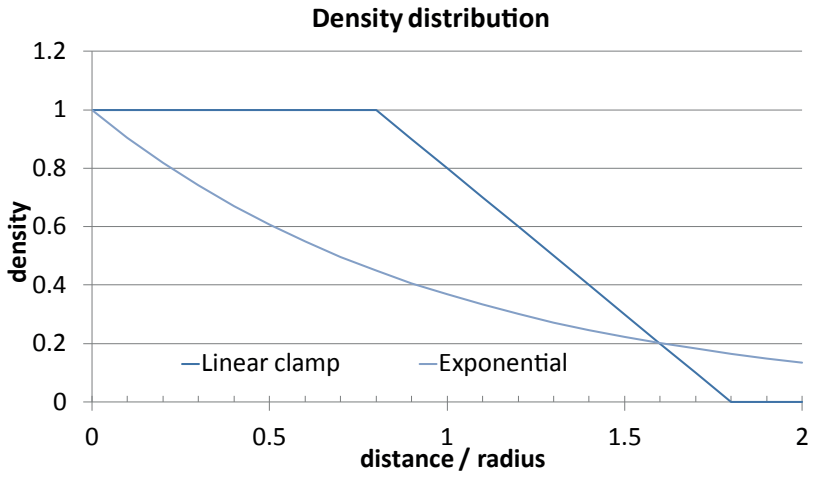


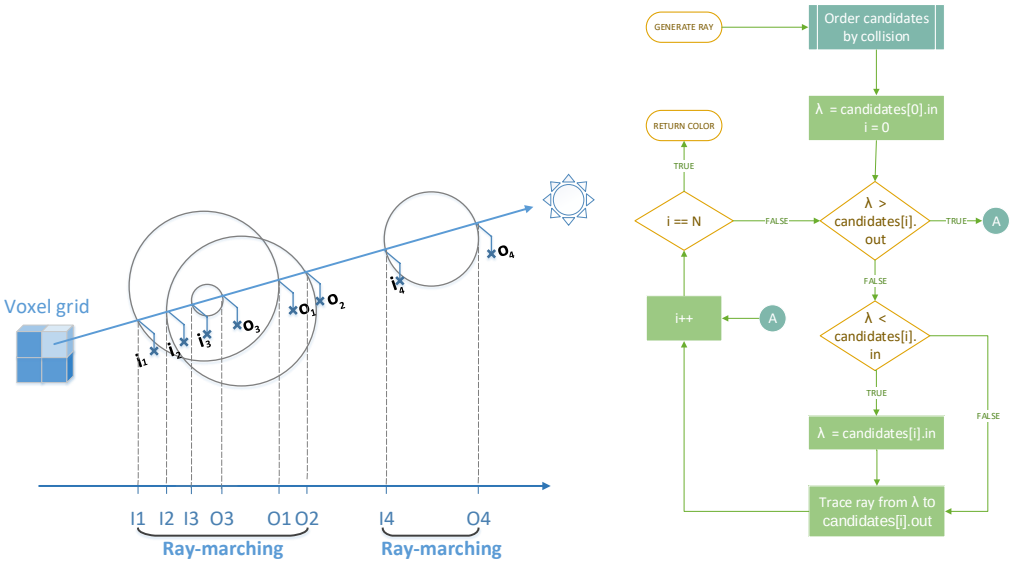

**Figure 4.** Distance/density relationship. The greater the distance between the ray and the sphere center, the lower the density of water vapour. The exponential approach gives natural realism by softening the cloud borders.

### *3.3. The No Duplicate Tracing Algorithm*

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 2. The image and flow chart in Figure 5a,b respectively explain the possible cases that may arise during the execution of a ray-marching.

**Algorithm 2** No-Duplicate-Tracing

```
candidates ← sortCandidates(C, n)
λ ← candidates[0]_{λ_in}
for j ← 0 < n do
    if λ > candidates[j]_{λ_out} then continue
    else
        if λ < candidates[j]_{λ_in} then
                λ = candidates[j]_{λ_in}
        end if
    end if
    while λ ≤ λ_out = candidates[j]_{λ_out} do
        {Compute transmittance}
        λ ← λ + Δ
    end while
end for
```




(**a**) (**b**)

**Figure 5.** Graphical and procedural explanation of *the no duplicate tracing algorithm*. (**a**) 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; (**b**) A flow chart illustrating the *the no duplicate tracing algorithm* (Algorithm 2).

Others research regarding bounding volumes overlapping is the work of Lipuš and Guid [31].

### *3.4. Scene Delimitation*

The explained technique eliminates the need to use complex space-partitioning algorithms like K-D trees, Octrees, or Quadtrees for locating primitives, due to the fact that each cloud only needs a small number of spheres to create a cumulus in our model. In addition, the use of bounding boxes to delimit the volume of each cloud also improves performance. The proposed system uses Smits' algorithm [7] and the improvement of Reference [32] to limit ray-marching to the volume inside the cloud's boundaries, as shown in Figure 6.

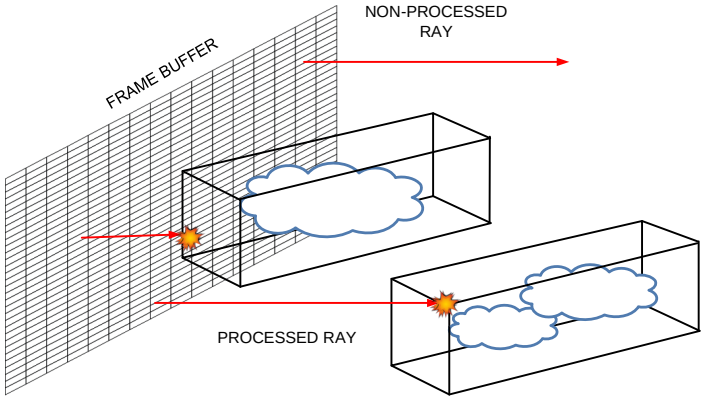



**Figure 6.** Bounding box delimitation of clouds. Smits' algorithm is useful for optimizing ray-marching by calculating the segment of the Euclidean straight line along which rendering must be performed according to the $\lambda$ value.

This method increases the frames-per-second (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 1 in Section 3.2. The improved Smits' algorithm is implemented in GPU code in a fragment shader with a very reduced linear cost.

### *3.5. Lighting and Shading*

To optimize performance, we used a two-pass drawing algorithm. The first pass computes 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 [12], and the code optimizations via the approximations of Harris [13] and Tessendorf [33] in our pseudo-spheroid-based volumetric rendering. The main differences with these works are the application of the ray-marching technique over a set of pseudo-spheroids/ellipsoids instead of a particle system, the use of a pre-calculated voxel grid with our NDT algorithm, and the combination of simplifications to generate a new model of code optimization. This is summarized in Algorithm 3.

**Algorithm 3** Lighting and Shading

**function** LIGHTING($\rho, rayPos, \vec{rayDir}, \vec{sunDir}, gridIndex, sunColor$)
  $\Delta_\tau \leftarrow e^{-\phi\rho}$
  {Calculate Voxel}
  $voxelIndex \leftarrow \frac{rayPos - gridMin}{gridMax - gridMin}$
  {Precomputed-light retrieval}
  $pL \leftarrow texture(gridIndex, voxelIndex)$
  {Absorbed-light calculation}
  $aL \leftarrow sunColor \circ pL$
  {Scattered-light calculation}
  $sL \leftarrow aL \cdot phase(g, \vec{rayDir}, \vec{sunDir}) \cdot \gamma$
  {Total-light calculation}
  $tL \leftarrow aL + sL$
  {Calculate cloud color}
  $color \leftarrow (1 - \Delta_\tau) \cdot tL \cdot \tau$
  $\tau \leftarrow \tau\Delta_\tau$
  **return** $(color, 1 - \tau)$
**end function**

The mentioned algorithms are illustrated in Figure 7 where the green ray represents the illumination pass pre-computed in the CPU and the red ray represents 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 contributions of emission and reflection. This part is executed in real-time in the GPU.

- Transmittance

  This calculation is performed in each pass. 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 of 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:

$$\tau(s) = \rho(s) \cdot A. \tag{3}$$

$$T(s, s') = e^{\int_s^{s'} \tau(t)dt}. \tag{4}$$

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 a particle projected on the plane perpendicular to the ray. For the forthcoming expressions we will use the simplified notation $T(s, s')$.

- Illumination 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 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 ray-marching 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 light incident on point $v$ is hence:

$$L(v) = \underbrace{I_0 \cdot T(0, D)}_{\text{absorption}} + \underbrace{\int_0^D C(l) \cdot T(0, l)dl}_{\text{scattering}}. \tag{5}$$

  The first term in Equation (5) is the light propagated across the volume of the cloud to point $v$, which is affected by the total transmittance from 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.

  The term $C(l)$ is the approximation of the forward light-scattering density according to Reference [13]. This work reports that 50% of the light scatters in the forward direction of the ray over a small solid angle $\gamma$ that is of about 0.0001 steradians. The remaining 50% is ignored because it requires a large amount of computation. Hence:

$$C(l) = \frac{1}{2}\frac{\gamma}{4\pi}. \tag{6}$$

  Equation (5) is converted to its discrete form and calculated using a ray-marching algorithm with step size $\Delta l$. Given $T_i$ is the transmittance between the target point and point $i$ along the ray, the incident light is:

$$L(\vec{v}) = \sum_{i=0}^{D/\Delta l} \frac{1}{2}\frac{\gamma}{4\pi} \cdot T_{i-1} + I_0 T_D. \tag{7}$$

- Render Pass (GPU side)

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

  The light reflected at point $s$ inside the cloud depends on the illumination of this point *L(s)* and the area occluded by particles $\tau(s)$. The reflected light is further affected by the transmittance from 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 a 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 hence expressed as:

$$I(\vec{x}) = \int_0^P \left[ \underbrace{L(s)\tau(s)}_{\text{reflection}} + \underbrace{L(s)\tau(s)P(\vec{n}_s, \vec{n}_l)}_{\text{scattering}} \right] \cdot T(0,s)ds. \tag{8}$$

The first term in Equation (8) 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 $\vec{n}_l$ refracted in the direction $\vec{n}_s$. Both terms are affected by the transmittance $T(0,S)$from the surface to point $s$.

To compute the emitted light using a ray-marching algorithm with step size $\Delta s$, we used the approximation proposed by Reference [33], that converts the continuous integral into a summation of segment integrals:

$$I(\vec{x}) = \sum_{i=0}^{P/\Delta s} \int_0^{\Delta s} L(i\Delta s + t)\left[1 + P(\vec{n}_s, \vec{n}_l)\right] \cdot \tau(i\Delta s + t) \cdot T(0, i\Delta s + t)dt. \tag{9}$$

Let $T_i$ represents the transmittance from the surface to point $i$. The phase term is considered constant along the segment integral and can be factored out of the integral:

$$I(\vec{x}) = \sum_{i=0}^{P/\Delta s} \left[T_i + T_i P(\vec{n}_s, \vec{n}_l)\right] \int_0^{\Delta s} L(i\Delta s + t)\tau(i\Delta s + t)dt. \tag{10}$$

The differential transmittance $\Delta T_i$ along segment $\Delta s$ is denoted as:

$$\Delta T_i = e^{-\int_0^{\Delta s} \tau(\vec{x} + s\vec{n})ds}. \tag{11}$$

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

$$\Delta T_i = 1 - \int_0^{\Delta s} \tau(\vec{x} + s\vec{n}_s)ds. \tag{12}$$

By employing this result and assuming that $L(i\Delta s + t)$ is constant inside the segment integral, Equation (10) is simplified to:

$$I(\vec{x}) = \sum_{i=0}^{P/\Delta s} \left[T_i + T_i P(\vec{n}_s, \vec{n}_l)\right] L_i \cdot (1 - \Delta T_i). \tag{13}$$

In our implementation, we employ the *Henyey–Greenstein* phase function to approximate the refraction of light in water droplets:

$$P(\vec{n}_s, \vec{n}_l) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g\vec{n}_s \cdot \vec{n}_l)^{3/2}} \tag{14}$$

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

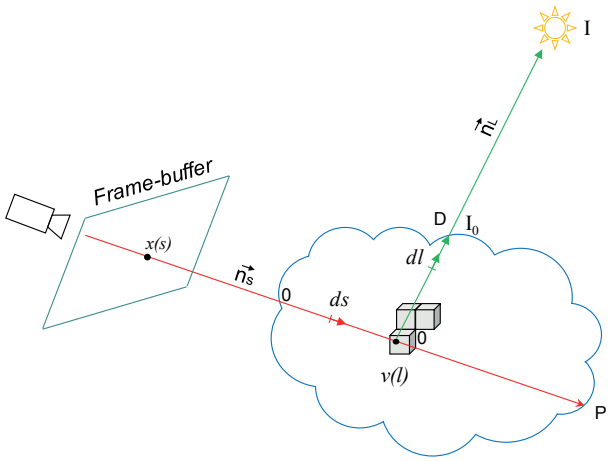



**Figure 7.** A two-pass algorithm for lighting along ray $l$ and rendering along ray s. $\vec{n_s}$ and $\vec{n_l}$ are the direction vectors of the ray-marching and the light respectively. $[0, P]$ and $[0, D]$ are the processing dimensions of line integral in the cloud for ray-marching and light respectively. $x(s)$ is frame-buffer pixel and $v(l)$ is a point into a voxel.

### *3.6. Cloud Shape Improvement*

This section presents four different methods to define the geometry of the cloud which fulfill different requirements from an artistic point of view. These methods range from purely mathematical random generation to user-defined 3D meshes modeled in an editor.

#### 3.6.1. 3D Gaussian Cloud Generation

The proposed algorithm accomplishes cumulus generation by using a three-dimensional Gaussian distribution of basic pseudo-spheroid primitives, as already discussed in Section 3.2. Our implementation uses a Gaussian distribution with clamping to generate the position of the pseudo-spheroids as is done for particles in the method by Huang et al. [1]. Unlike their work, the number of primitives to produce a realistic cloud is much smaller than in a particle system. Typically we use 35 pseudo-spheroids while the particle system requires thousands of them.

Let P be the position of the pseudo-spheroid, C the center of the cloud, and $\mathcal{N}$ a random variable with normal distribution, the base equation is then:

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

Table 2 shows the typical values, $\mu$ (mean) and $\sigma$ (standard-deviation), of the normal distribution used for each of the axes. The previous values must be added to the center of the cumulus to place the cloud into the scene.

**Table 2.** Typical parameters for our Gaussian equations where $k_i$, $t$ and $m_i$ are scale constants.

| Axis | $\mu$ | $\sigma$ | Clamped |
|---|---|---|---|
| X | $\mu_x$ | $\sigma_x$ | $[-k_1\sigma_x, k_2\sigma_x]$ |
| Y | $\mu_y$ | $\sigma_y$ | $[\mu_y, t\sigma_y]$ |
| Z | $\mu_z$ | $\sigma_z$ | $[-m_1\sigma_z, m_2\sigma_z]$ |

The radii of the pseudo-spheroids depend on the distance from their center (P) to the center of the cloud (C) according to a Gaussian function, as shown in Figure 8. Two variants of the proposed distribution can be used to compute the magnitude of the radius as described in Equations (15) and (16):

$$radius_i = \frac{\varepsilon_1}{|P_x| + |P_z| + 1.0} \tag{15}$$

$$radius_i = \varepsilon_2 \cdot \left(1.0 - 0.1\sqrt{((P_x - C_x)/2\sigma_x)^2 \cdot ((P_y - C_y)/2\sigma_y)^2 \cdot ((P_z - C_z)/2\sigma_z)^2}\right), \tag{16}$$

where $\varepsilon_i$ 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 as seen in Figures 9 and 10 generated with Equation 16.

We introduced 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 ∼20% of spheroids in the nucleus of the cloud, which yields a ∼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 $(x_j, y_j, z_j)$:

$$\forall i \forall j (r_i - r_j) \geq \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}. \tag{17}$$

The $O(n^2)$ complexity of the previous computation is not a drawback due to the reduced number of spheres required in our Gaussian model.

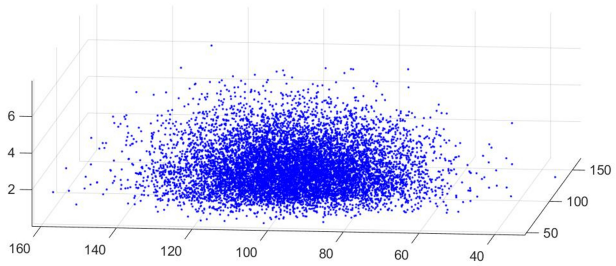



**Figure 8.** A three-dimensional Gaussian distribution. Some cloud shapes resemble a 3D Gaussian normal distribution, specifically the cumuliform ones. The cause is the altitude at which the condensation is produced. Since the condensation level may be conceived as a flat and horizontal surface, the clouds formed at this level have a flat bottom side, as cited by Häckel's book [34].

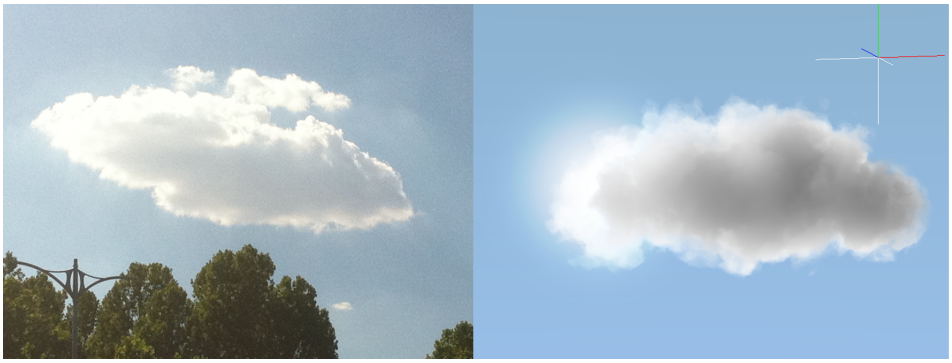


**Figure 9.** Example of clouds with Gaussian distribution. A real photograph (**left**) and our generated cumulus (**right**) using 35 pseudo-spheroids in total.

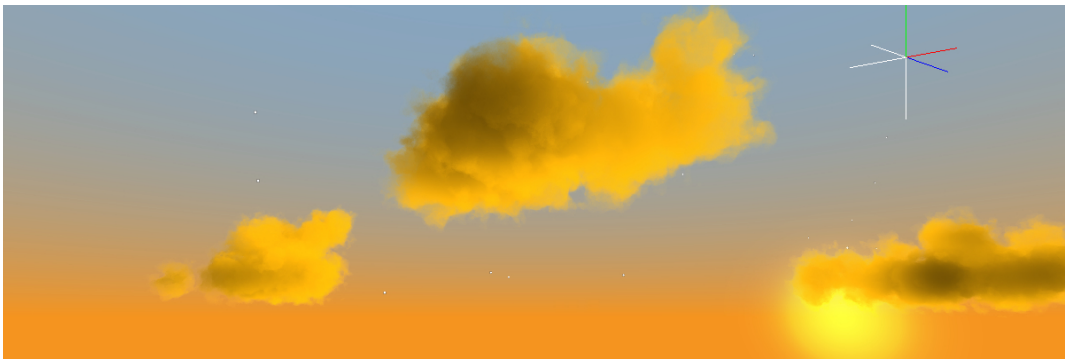


**Figure 10.** A crepuscular landscape with three Gaussian cumuli executing the volumetric scattering and voxel shading with cloud interaction and occlusion.

#### 3.6.2. 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 formed [35]. 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 L-systems is given in Reference [36] with exhaustive algorithms for clouds and plants.

Our approach differs from that of Kang et al. [8]; in our approach, we include a proportional random factor in the generation of the sphere radius and the distance between primitives in each recursive call, as explained in Figure 11. Another difference from the cited work is the use of volumetric rendering and our own density function, Equation (2). The 3D algorithm for L-system generation of spheres is referred to on the website of the *Department of Computing for Neurobiology at Cornell University*, which contains a complete MATLAB explanation and sources. Before calling the GLSL image renderer, 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 an axiom and rules. For example, in the call to the following Backus–Naur form (BNF) grammar, our C++ algorithm will generate a string of "*X*", "*Y*", "+" and "&" symbols:

$$\begin{aligned} Axiom &\rightarrow X \\ X &\rightarrow X + Y \\ Y &\rightarrow X \,\&\, Y, \end{aligned} \tag{18}$$

where "+" means turn left with angle $\delta$ using a rotation matrix, and "&" means pitch down with angle $\delta$ using a different rotation matrix.

As a result, Figure 12a,b were generated using different angles and iterations. The corresponding lexical analyser must insert a sphere of a 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 (18) 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.

Our L-system grammar derivative is suitable for generating jet streams and a sort of cloud called *Castellanus*, as cited by Häckel [34] 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.

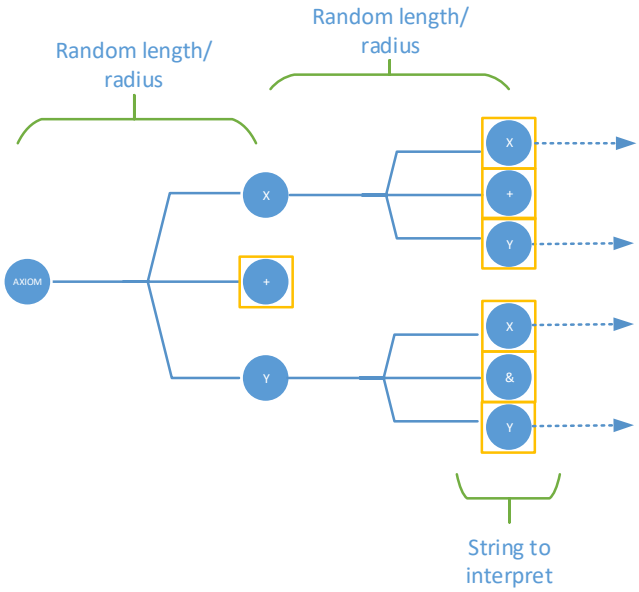
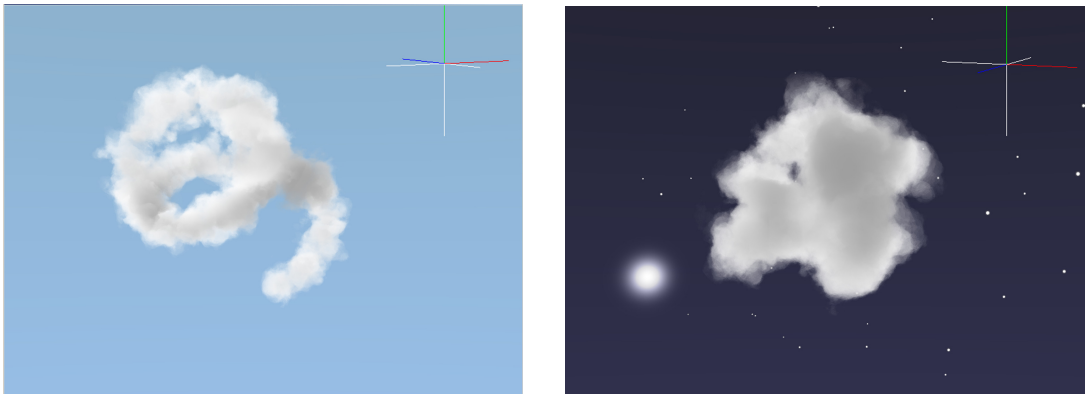



**Figure 11.** After each recursive call, the interpreter generates a new proportional random radius and length for the primitives. This produces more natural and impressive cumuliforms.



(**a**) (**b**)

**Figure 12.** Generated clouds from the previous L-system grammar derivation. (**a**) Iterations = 7, $\delta = 10°$. The lower the angle $\delta$ , the thinner and more strange the cloud shape becomes, hence 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; (**b**) Using exactly the same grammar and iterations with a larger angle, for instance $[50°, 100°]$, the resulting grammar derivation generates dense cumulus formations.

### 3.6.3. Cumuliforms with Optimized Metaballs

The metaball technique proposed by Blinn [37] for the creation of organic-looking n-dimensional objects can also be applied to generate cumulus formations with very high performance and render quality. Our model differs from the Dobashi approach [38] in that we use optimized equations and a mean calculated in the CPU:

$$f(x,y,z) = \gamma = \sum_{i=1}^{spheres} \frac{{r_i}^2}{(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2} \geq 1, \tag{19}$$

where $x, y, z$ is the ray-marching position, $r_i$ is the current sphere radius, and $x_i, y_i, z_i$ is the sphere center. Additionally we have:

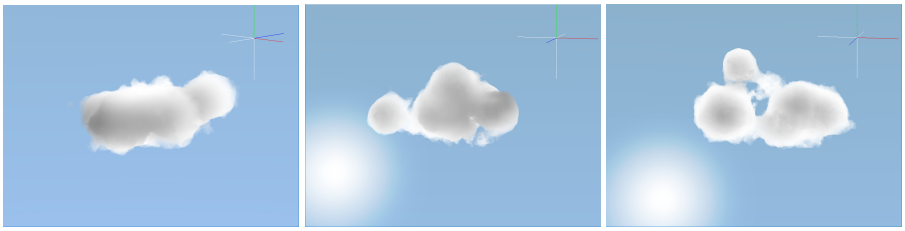
$$\bar{r}^2 = \sum_{i=1}^{N=spheres} \frac{r_i{}^2}{N}, \tag{20}$$

and:

$$\rho = fbm(rayPos) \cdot \bar{r}^2 \cdot \phi. \tag{21}$$

Equation (20) is pre-calculated in the CPU only once, and Equations (19) and (21) are used in the GPU shader algorithm. $\phi$ 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 3.1. It usually has a value of 0.6 in our tests.

Our method has very high performance and achieves a remarkable number of frames-per-second. Besides this, render quality is sufficient, as shown in Figure 13, but lower than that generated by Algorithm 1.



(**a**) (**b**) (**c**)

**Figure 13.** Different cumulus formations using a Gaussian distribution and optimized metaballs. Just six spheres were used to render the samples by randomizing the sphere radius.

#### 3.6.4. Cloud Generation from 3D Meshes

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 general public license (GPL) *Blender* application as a geometry editor, and this was employed for several models used in this research. Effective real-time rendering needs a prior decimation of big meshes down 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 our 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 techniques explained in Sections 3.1–3.5. 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 radii (a, b, c). This approach is effective and accurate, as explained in Figure 14.

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

$$B = \Big( \sum_{i=1}^{3} \frac{x_i}{3}, \sum_{i=1}^{3} \frac{y_i}{3}, \sum_{i=1}^{3} \frac{z_i}{3} \Big), \tag{22}$$

and the scale as:

$$P_i' = (P_i - B) \cdot scale + B \tag{23}$$

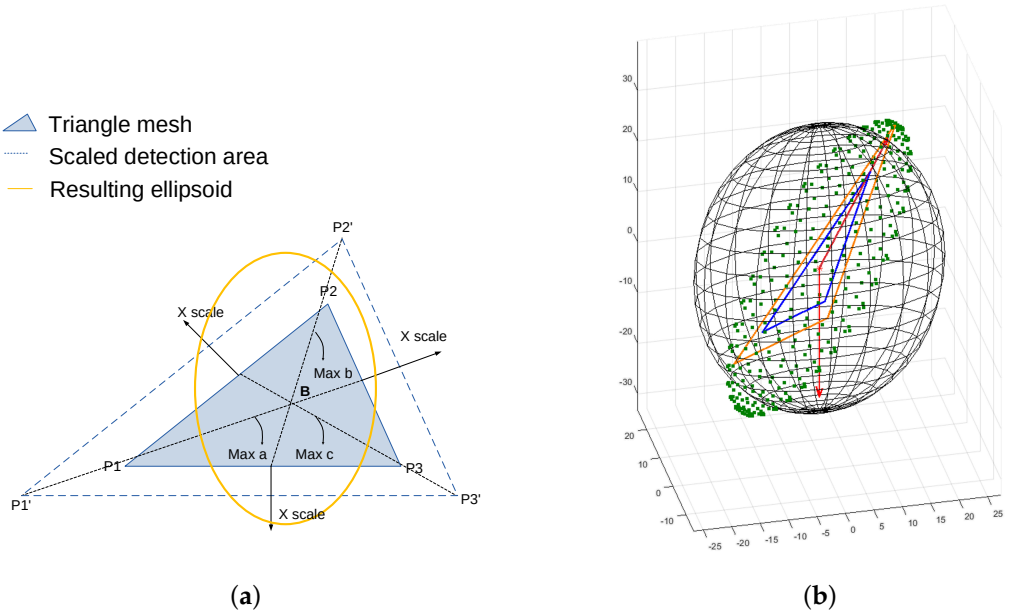
$$\begin{aligned} radius_a &= \|(B - P_1')\| \\ radius_b &= \|(B - P_2')\| \\ radius_c &= \|(B - P_3')\|, \end{aligned} \tag{24}$$

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

An optimization of the previous algorithm can be made by performing an $R^3$ rotation of the direction vector, where the maximum radius of the ellipsoid is chosen to align with the maximum distance from the triangle vertices to the barycenter. The solution requires the application of the principles of *Rodrigues' Rotation Formula*. 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 results. Therefore, continuations of the previous formulas are:

$$if \begin{cases} max(radius_a) & \vec{dirTria} = P'_1 - B \\ max(radius_b) & \vec{dirTria} = P'_2 - B \\ max(radius_c) & \vec{dirTria} = P'_3 - B \end{cases} \tag{25}$$

$$if \begin{cases} max(radius_a) & \vec{dirEllip} = (radius_a + B_x, B_y, B_z) - B \\ max(radius_b) & \vec{dirEllip} = (B_x, radius_b + B_y, B_z) - B \\ max(radius_c) & \vec{dirEllip} = (B_x, B_y, radius_c + B_z) - B \end{cases} \tag{26}$$




(**a**) (**b**)

**Figure 14.** (**a**) 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 to estimate the density in the ellipsoid/ray collision equations; (**b**) After rotation. As seen in the image above, showing the original ellipsoid in black and the resulting one in green, the previous equations allow overlapping of the $R^3$ direction vectors. Hence, according to the direction of the larger triangle vertex, the algorithms produce the resulting rotation.

The rotation is hence calculated via the following steps.
(1) Calculate the axis and angle using cross products and dot products:

$$x = \frac{\vec{dirEllip} \times \vec{dirTria}}{\|\vec{dirEllip} \times \vec{dirTria}\|} \tag{27}$$

$$\theta = cos^{-1}\left(\frac{\vec{dirEllip} \cdot \vec{dirTria}}{\|\vec{dirEllip}\| \cdot \|\vec{dirTria}\|}\right). \tag{28}$$

(2) Calculate the rotation matrix using an exponential map:

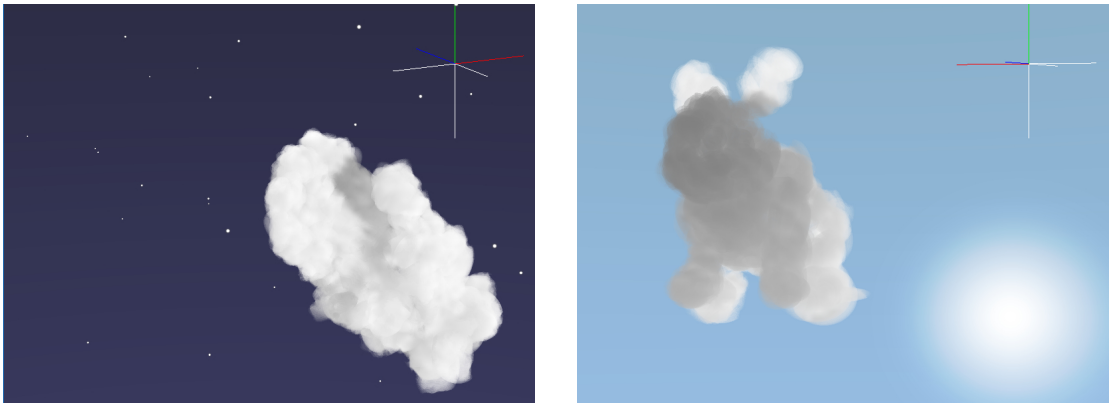
$$R = e^{A\theta} = I + sin(\theta) \cdot A + (1 - cos(\theta)) \cdot A^2. \tag{29}$$

(3) Calculate $A$, a skew-symmetric matrix corresponding to $x$:

$$A = [x]_x = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}. \tag{30}$$

Formulas (1)–(3) are pre-calculated in the CPU side only once, and only the rotation matrix $R$ is passed to the GPU for real-time rendering. Thanks to this algorithm, the overall performance is not hindered even in lower speed GPUs as it is demonstrated in the benchmarks Section 4.2.

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 15. A 95% decimation still produces good visualization and performance. Lower decimations may be considered depending on the hardware being used.



(**a**) (**b**)

**Figure 15.** Example of 3D meshes converted to cumuliform clouds resembling known shapes. (**a**) A hand mesh transformed into a soft 3D cloud. The final result is successfully optimized for real-time GPU algorithm animation and rendering; (**b**) 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.

## 4. Results

To confirm the initial hypothesis, benchmarks and measurements were taken using different GPUs to evaluate the balance between performance and realism. Analytical verification of Algorithm 1 that runs on the GPU was also performed to assess its complexity.

### *4.1. Complexity Analysis*

Let $|B|$ be the number of clouds, $n$ be the bounding box near-plane resolution area in pixels, $s$ be the number of pseudo-spheroids in the bounding box, $c$ be the size of selected candidates to be rendered, and $d$ be the depth of ray-marching. Then, Algorithm 1 has the following execution times (numeric constants refer to the instruction counts):

- 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:

$$t_{best-best} = \sum_{i=1}^{|B|}\sum_{j=1}^{n}\left(1 + \underbrace{\sum_{t=1}^{s}(1+6)}_{getCandidates()}\right) = \Omega(ns). \tag{31}$$

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

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

- 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:

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

- 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:

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

Our 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 $O(nlog_2 n)$ instead of $O(n^2)$ scaling, expecting that it would provide better performance. However, no improvement in frames-per-second (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 our in-line optimizations we could not exceed the speed of *insertion sort*. Another alternative for the *insertion sort algorithm* is *Batcher's Odd–Even Mergesort* as cited by Kipfer and Westermann [39], due to the better low-level parallelization and its worst-case complexity parallel time, represented by $O(log^2(n))$.

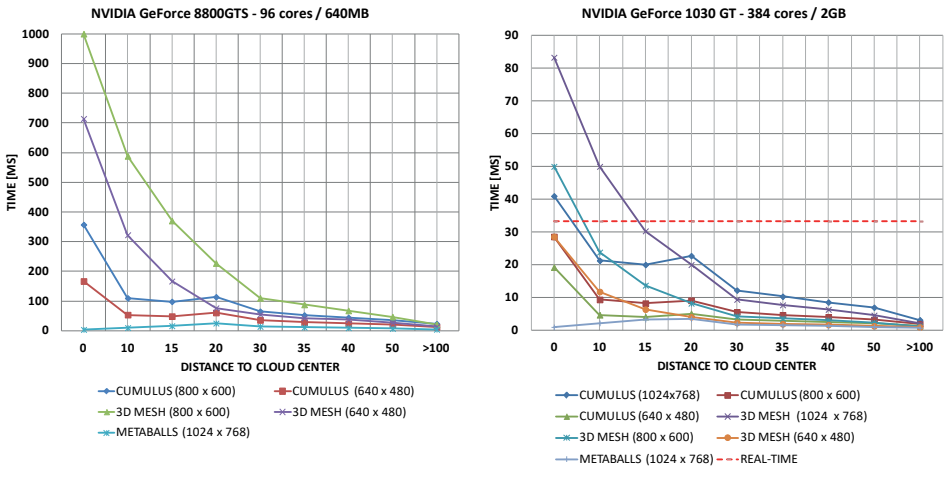
### *4.2. Benchmark Tests*

A set of tests were 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 (Pascal, 640 cores) and a GeForce GTX 970 (1664 cores), running on a 64-bit i-Core 7 CPU 860@2.80 GHz (first generation, 2009) with 6 GB random access memory (RAM), see Figure 16. The project was implemented entirely in C++ using the OpenGL and GLM math libraries for the host side and the OpenGL Shading Language (GLSL) for the GPU side. The ray-marching step size was determined by $\lambda$ in line 50 of Algorithm 1 for the cumulus test and was set to be the constant value 0.1 for the 3D mesh tests. Promising results were obtained when the cloud was far from the camera with the 8800 GTS at $800 \times 600$ and $640 \times 480$ pixels, especially with metaballs.

The same algorithm suite performs perfectly in all resolutions using the nVidia GTX 970, GT 1030, and GTX 1050, 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 Reference [40] for cumulus rendering. For this benchmark, we used 35 spheres for cumulus generation with the filter explained in Section 3.6.1 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 (NDT) algorithm. Without the NDT algorithm, the CPU execution time would double 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 $1920 \times 1080$ 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%. Table 3 summarizes the minimum distance in Full-HD ($1920 \times 1080$ pixels) where the analysed graphic cards reach real-time with our algorithms.

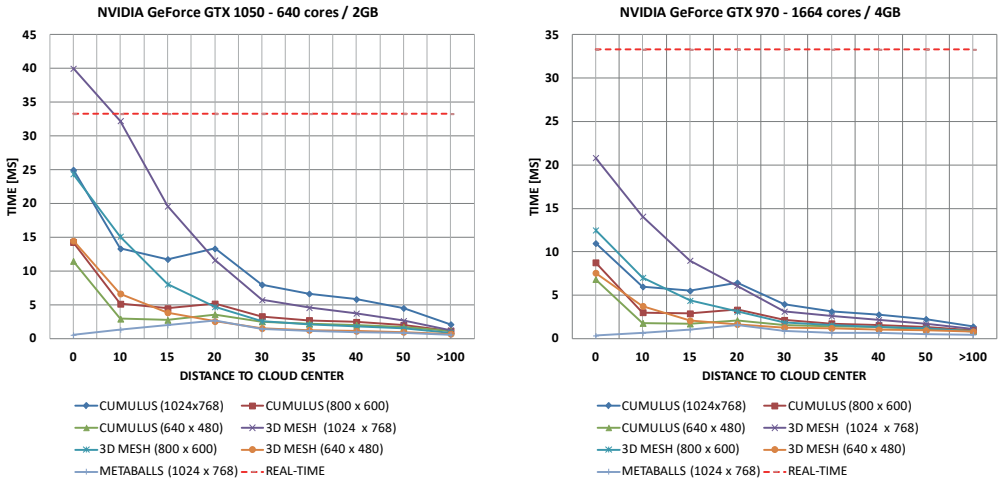
**Table 3.** The table below shows the minimum distance from the cloud at which full high-definition (HD), $1920 \times 1080$ pixels, rendering reaches 30 frames-per-second (FPS, minimum real-time). This distance is suitable for scenarios where getting close to the surface of the cloud is required.

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




(**a**) (**b**)

**Figure 16.** *Cont.*




(**c**) (**d**)

**Figure 16.** The performance of the algorithms on different graphic cards. (**a**) With the GeForce 8800 GTS, the performance at 800 × 600 pixels overcomes the limit of the hyper-realistic method shown in Reference [40]; (**b**) 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; (**c**) Performance in a GeForce GTX 1050 is optimum in 99% of cases; (**d**) 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.

## 5. Discussions

In comparison with particle system and basic ray-tracing methods, the proposed algorithm results in greater realism than that found in References [1,8,14,41–43], as shown in Figure 17a–c, and higher performance, as shown in Table 4. The obtained realism is comparable to that of off-line and photo-realistic models that use all physical characteristics, as shown in Figure 18, which are not suitable for use in real time due to the long execution times required. A novel non-real time approach is found in Reference [2] which applies the radiance-predicting neural network model (RPNN) to emulate real cumuli. However, it takes ∼12 h 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 overcome the limitations of the non-real time and photo-realistic methods cited in References [44–46]. It is also a suitable framework for implementing lighting and shadowing algorithms, with lower GPU overhead as compared to the methods of other works [47–49]. Our research improves pre-computation times to the millisecond level as opposed to minutes [19,48,50], or hours [2].

With respect to the research by Bouthors [40], our model shows an increase in FPS performance on the same graphics hardware. While not obtaining the hyper-realism of radiometry achieved in that model, our cloud generation method using 3D meshes improves on the geometric accuracy and smoothness obtained by Wither et al. in the rapid sketch modelling of clouds [51]. In contrast to the realistic method by Mukhina [4], where the cloud is projected on an hemispherical disc, our volumetric algorithm allows for the navigation and traversal of gaseous mass to observe details. Our method also improves on the overall realism in the model of Elek et al. [20], while maintaining the same performance as the conservative ray-marching version of their implementation when executed at 1920 × 1080 pixels. Despite our efforts, our method lacks the precision of the excellent work by Klehm et al. [23] and Peters et al. [24] with respect to scattering and shadow maps.

None of the algorithms cited above cover the aspect of cloud motion and shape alteration. Our algorithm does not cover this aspect either, but it is well-suited to implement the effect of wind advection over a dynamic system of primitives.

(**a**) (**b**) (**c**)

**Figure 17.** Differences between our model and other works. (**a**) A generated cloud using the particle systems of Harris [13], and Huang et al. [1]. The contour of the cloud and the overall realism lack accuracy. (**b**) The cloud modeling method of Montenegro et al. [41] that combines procedural and implicit models. (**c**) Our ontological volumetric cloud rendering method with lighting. The procedural noise improves cloud edges and fuzzy volume effects.

**Table 4.** Average frequency of other particle and volumetric systems compared with our method.

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


**Figure 18.** Photo-realistic clouds with all physical characteristics. The image above was generated with POV-Ray in 1 h, 18 min using 100% of the CPU cores at 1024 × 768 pixel resolution. 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 paper.

## 6. Conclusions and Future Work

We have presented a real-time cloud rendering method with a good balance between realism and performance. The algorithm uses flat uniform noise through fBm with low memory usage ($64 \times 64 \times 64$ single-precision floats) to ensure efficient computational costs of ray-marching. In addition, the use of bounding boxes with very few spheroids allows for the application of a simple linear loop which discards clouds outside the camera view. This low number of spheroids is achieved by a new 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 longer coding time and complex CPU/GPU coordination.

The use of pre-calculated light stored in a voxel grid and optimizations like the *no duplicate tracing* algorithm improve the computing speed for light. 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 that multiple scattering is only calculated over a small solid angle, neglecting 50% of the remaining scattered light.

The algorithm implements a function that allows the use of 3D meshes to define the shapes of clouds that could resemble known 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.

In summary, taking into account the data presented in the results section, we can convey that the initial hypothesis is confirmed and this algorithm is a good candidate for applications requiring a balance between performance and realism, like computer games, flight simulators, and virtual reality.

There is still room for future improvements, for instance, the ontological implementation of cloud dynamics caused by wind advection. Such an implementation should act on individual primitives, changing size and position to reproduce the effects of cloud motion and shape evolution. The Supplementary material illustrates an mpeg-4 video with images regarding the different methods and algorithms explained in this paper.

**Supplementary Materials:** The following are available online at http://www.isometrica.net/symmetry-284454.mp4.

**Acknowledgments:** No grants were received to support this research and no funds were collected to cover the costs of publishing in open access. I would like to acknowledge the gentle guidance of Jacobo Rodriguez Villar at BlitWorks, and the Khronos Group from Asturias (Spain) with respect to GLSL and OpenGL API. I am also grateful to my brother, Antonio Jiménez de Parga, for his support and guidance. Also, I would like to express my unconditional gratitude to the National University of Distance Education for allowing my learning and improving my axiomatic self-taught work with respect to a multimedia native high performance C++ application developer. Finally, the authors would like to acknowledge the anonymous reviewers for their valuable comments and suggestions.

**Author Contributions:** Carlos Jiménez de Parga is the main author, the subject researcher, and the article writer. Sebastián Rubén Gómez Palomo is the thesis advisor and article reviewer.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript (alphabetically ordered):

| | |
|---|---|
| BNF | Backus-Naur form |
| CPU | Central Processing Unit |
| CT | Computed Tomography |
| fBm | Fractal Brownian Motion |
| FPS | Frames-per-second |
| GPL | General Public License |
| GPU | Graphics Processing Unit |
| LOD | Level of Detail |
| MRI | Magnetic Resonance Imaging |
| MS | Milliseconds |
| NDT | No Duplicate Tracing |
| RAM | Random Access Memory |
| RPNN | Radiance-predicting neural network |

## References

1. Huang, B.; Chen, J.; Wan, W.; Bin, C.; Yao, J. Study and Implement About Rendering of Clouds in Flight Simulation. In Proceedings of the International Conference on Audio, Language and Image Processing (ICALIP 2008), Shanghai, China, 7–9 July 2008; pp. 1250–1254.
2. Kallwet, S.; Müller, T.; McWilliams, B.; Gross, M.; Novák, J. Deep Scattering: Rendering Atmospheric Clouds with Radiance-Predicting Neural Networks. *ACM Trans. Graph.* **2017**, *36*, 231. [CrossRef]
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, Los Angeles, CA, USA, 30–31 July 2005; p. 9.

4. Mukina, K.; Bezgodov, A. The Method for Real-time Cloud Rendering. *Procedia Comput. Sci.* **2015**, *66*, 697–704. [CrossRef]
5. Arthur, A. Some techniques for shading machine renderings of solids. In Proceedings of the AFIPS '68 (Spring), Atlantic City, NJ, USA, 30 April–2 May 1968; Voume 32, pp. 37–45.
6. Whitted, T. An improved illumination model for shaded display. In Proceedings of the SIGGRAPH '79, Chicago, IL, USA, 8–10 August 1979; Voume 13, p. 14.
7. Smits, B. Efficient bounding box intersection. *Ray Tracing News* **2002**, *15*, 1.
8. Kang, S.Y.; Park, K.C.; Kim, K.I. Real-Time Cloud modelling and Rendering Approach Based on L-system for Flight Simulation. *Int. J. Multimed. Ubiquitous Eng.* **2015**, *10*, 395–406. [CrossRef]
9. Miyazaki, R.; Yoshida, S.; Dobashi, Y.; Nishita, T. A method for modelling clouds based on atmospheric fluid dynamics. In Proceedings of the Ninth Pacific Conference on Computer Graphics and Applications, Tokyo, Japan, 16–18 October 2001; pp. 363–372.
10. Gardner, Y.G. Visual Simulation of Clouds. In Proceedings of the ACM SIGGRAPH Computer Graphics, San Francisco, CA, USA, 22–26 July 1985; Volume 19, pp. 297–304.
11. Max, N. Computer Animation of Clouds. In Proceedings of the Computer Animation '94, Geneva, Switzerland, 25–28 May 1994; pp. 167–174.
12. Max, N. Optical Models for Direct Volume Rendering. *IEEE Trans. Vis. Comput. Graph.* **1995**, *1*, 99–108. [CrossRef]
13. Harris, M.J.; Lastra, A. Real-time Cloud Rendering. In *Computer Graphics Forum*; Blackwell Publishers: Oxford, UK, 2001; Volume 20, pp. 76–85.
14. Harris, M.J. Real-time Cloud Rendering for Games. In Proceedings of the Game Developers Conference, San Jose, CA, USA, 8–12 March 2002; pp. 1–14.
15. Ebert, D. Volumetric modelling with implicit functions (A cloud is born). In Proceedings of the ACM SIGGRAPH 97 Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH '97, Los Angeles, CA, USA, 3–8 August 1997; p. 147.
16. Engel, W. *Shader X5: Advanced Rendering Techniques (Shaderx)*; Charles River Media: Newton, MA, USA, 2007.
17. Drebin, R.; Carpenter, L.; Hanrahan, P. Volume rendering. In Proceedings of the SIGGRAPH '88, Atlanta, GA, USA, 1–5 August 1988; Volume 22, pp. 65–74.
18. Levoy, M. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.* **1988**, *8*, 29–37. [CrossRef]
19. Yusov, E. High-Performance Rendering of Realistic Cumulus Clouds Using Pre-computed Lighting. In Proceedings of the Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics, Los Angeles, CA, USA, 8–9 August 2014; pp. 127–136.
20. Elek, O.; Ritschel, T.; Wilkie, A.; Seidel, H. Interactive cloud rendering using temporally coherent photon mapping. *Comput. Graph.* **2012**, *36*, 1109–1118. [CrossRef]
21. Dobashi, Y.; Kaneda, K.; Yamashita, H.; Okita, T.; Nishita, T. A Simple, Efficient Method for Realistic Animation of Clouds. In Proceedings of the SIGGRAPH '00, New Orleans, LA, USA, 23–28 July 2000; pp. 19–28.
22. Goswami, P.; Neyret, F. Real-time Landscape-size Convective Clouds Simulation and Rendering. In Proceedings of the Workshop on Virtual Reality Interaction and Physical Simulation, Lyon, France, 23–24 April 2017; Fabrice, J., Florence, Z., Eds.; The Eurographics Association: Munich, Germany, 2017.
23. Klehm, O.; Seidel, H.; Eisemann, E. Prefiltered Single Scattering. In Proceedings of the ACM I3D, San Francisco, CA, USA, 14–16 March 2014; pp. 71–78.
24. Peters, C.; Münstermann, C.; Wetzstein, N.; Reinhard, K. Beyond hard shadows: Moment shadow maps for single scattering, soft shadows and translucent occluders. In Proceedings of the ACM I3D, Redmond, WA, USA, 26–28 February 2016; pp. 159–170.
25. Perlin, K.; An Image Synthesizer. *ACM SIGGRAPH Comput. Graph.* **1985**, *19*, 287–296. [CrossRef]
26. Hassan, M.M.; Sazzad, K.M.; Ahmed, E. Generating and Rendering Procedural Clouds in Real Time on Programmable 3D Graphics Hardware. In Proceedings of the 9th International Multitopic Conference (IEEE INMIC 2005), Karachi, Pakistan, 24–25 December 2005; pp. 1–6.
27. Apodaca, A.; Gritz, L. *Advanced Renderman. Creating CGI for Motion Pictures*; Morgan Kaufmann: San Francisco, CA, USA, 2000.
28. Elinas, P.; Stuerzlinger, W. Real-time rendering of 3D Clouds. *J. Graph. Tools* **2000**, *5*, 33–45. [CrossRef]
29. Shirley, P.; Keith, R. *Realistic Ray-Tracing*, 2nd ed.; A K Peters: Wellesley, MA, USA, 2003.

30. Gil, B. *Mathematics I*; Edelvives Publishers: Zaragoza, Spain, 1988.
31. Lipuš, B.; Guid, N. A new implicit blending technique for volumetric modelling. *Vis. Comput.* **2005**, *21*, 83–91. [CrossRef]
32. Williams, A.; Barrus, S.; Keith, R.; Shirley, P. An efficient and robust ray-box intersection algorithm. In Proceedings of the SIGGRAPH '05 ACM SIGGRAPH 2005, Los Angeles, CA, USA, 31 July–4 August 2005; pp. 1–4.
33. Tessendorf, J. *Resolution Indepent Volumes*; School of Computing, Clemson University: Clemson, NC, USA, 2016.
34. Häckel, H. *Wolken*; Ulmer Eugen Verlag Publishers: Stuttgart, Germany, 2004.
35. Prusinkiewicz, P.; Lindenmayer, A. *The Algorithmic Beauty of Plants*; Springer: Berlin, Germany, 1996.
36. Jones, H. *Computer Graphics through Key Mathematics*; Springer: Berlin, Germany, 2001.
37. Blinn, J.F. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph.* **1982**, *1*, 235–256. [CrossRef]
38. Dobashi, Y.; Nishita, T.; Yamashita, H.; Okita, T. Using metaballs to modelling and animate clouds from satellite images. *Vis. Comput.* **1999**, *15*, 471–482. [CrossRef]
39. Matt, P.; Randima, F. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*; Addison-Wesley Educational Publishers: Boston, MA, USA, 2005.
40. Bouthors, A.; Neyret, F.; Max, N.; Bruneton, E.; Crassin, C. Interactive multiple anisotropic scattering in clouds. *Int. J. Multimed. Ubiquitous Eng.* **2008**, 173–182.
41. Montenegro, A.; Baptista, I.; Dembogurski, B.; Clua, E. A new method for modelling clouds combining procedural and implicit models. In Proceedings of the Simpósio Brasileiro de Jogos e Entretenimento Digital, Curitiba, Brazil, 1–4 November 2017.
42. Bi, S.; Bi, S.; Zeng, X.; Lu, Y.; Zhou, H. 3-Dimensional modelling and Simulation of the Cloud Based on Cellular Automata and Particle. *ISPRS Int. J. Geo-Inf.* **2016**, *5*, 86. [CrossRef]
43. Horng-Shyang, L.; Tan-Chi, H.; Jung-Hong, C.; Cheng-Chung, L. Fast rendering of dynamic clouds. *Comput. Graph.* **2005**, *29*, 29–40.
44. Narasimhan, S.; Grupta, M.; Donner, C.; Ramamoorthi, R.; Nayar, S.; Jensen, H. Acquiring Scattering Properties of Participating Media by Dilution. *ACM Trans. Graph.* **2006**, *25*, 1003–1012. [CrossRef]
45. Jarosz, W.; Donner, C.; Zwicker, M.; Jensen, H. Radiance Caching for Participating Media. *ACM Trans. Graph.* **2008**, *27*. [CrossRef]
46. Cerezo, E.; Perez, F.; Pueyo, X.; Seron, F.; Sillion, F. *A Survey on Participating Media Rendering Techniques*; Springer: Berlin, Germany, 2005; Volume 21, pp. 303–328.
47. Gautron, P.; Marvie, J.; Francois, G. Volumetric Shadow Mapping. In Proceedings of the SIGGRAPH '09, Yokohama, Japan, 16–19 December 2009.
48. Zhou, K.; Ren, Z.; Lin, S.; Bao, H.; Guo, B.; Shum, H. Real-time smoke rendering using compensated ray marching. In Proceedings of the SIGGRAPH '08, Los Angeles, CA, USA, 11–15 August 2008; pp. 1–12.
49. Delalandre, C.; Gautron, P.; Marvie, J.; Francois, G. Single scattering in heterogeneous participating media. In Proceedings of the SIGGRAPH '10, Los Angeles, CA, USA, 26–30 July 2010; p. 1.
50. Ament, M.; Sadlo, F.; Weiskopf, D. Ambient Volume Scattering. *IEEE Trans. Vis. Comput. Graph.* **2013**, *19*, 2936–2945. [CrossRef]
51. Wither, J.; Bouthors, A.; Can, M. Rapid sketch modelling of clouds. In Proceedings of the Eurographics Workshop on Sketch-Based Interfaces and modelling (SBIM), Annecy, France, 11–13 June 2008; pp. 113–118.