Diplomarbeit

A Real-Time Cloud Animation and Illumination Method

ausgeführt am
Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
und
Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
als verantwortlich mitwirkendem Universitätsassistenten

durch

Robert Fizimayer
Matrikelnummer 0025821
Eisenstädter Strasse 100
A-7091 Breitenbrunn

....
Unterschrift Datum

Abstract

Over the last years, real-time rendering, especially for virtual environments as can be seen in most computer games, became more and more important. As the capabilities of graphics hardware continuously increases, the expectations of users regarding to visual quality are also increasing relatively fast.

Compared to the rapid development in many fields of computer graphics, a very important element for real-time outdoor scene rendering has been almost completely disregarded. Clouds can highly improve the visual quality of a scene due to their complex structure, the beautiful colors as well as their formation and movement. Because of the computational effort to render animated and dynamically lit clouds, it has been impossible for consumer graphics adapters to draw them and simultaneously keep the framerate high enough. Although there are many methods which allow drawing clouds in outdoor environments fast enough, all of them have some restrictions either in movement, lighting or formation.

This thesis will examine how plausible cloud animation can be done in real time on consumer graphics hardware and how the animation results can be used to render high-quality, dynamically lit, volumetric clouds with relatively high frame rates. Since high-quality solutions already exist for non-real-time rendering applications and they are almost completely computed on the CPU, which is not acceptable for most games since the CPU is very often already at peak load, methods will be discussed to adapt those algorithms to be performed on the GPU, while also some new improvements will be presented.

The thesis first gives an overview of state-of-the-art of cloud rendering algorithms for real-time as well as for non-real-time rendering. Then, we will try to adapt parts of these techniques to work in interactive environments at high framerates while all significant characteristics of the non-real-time methods like cloud formation, extinction, movement and dynamic lighting will still be preserved.

Kurzfassung

In den letzten Jahren gewann das Gebiet des Echtzeit-Renderings, insbesondere im Bereich der virtuellen Umgebungen wie sie in den meisten Computerspielen zu finden sind, zunehmend an Bedeutung. Mit der steigenden Leistungsfähigkeit der Grafikbeschleuniger stiegen auch die Erwartungen der Benutzer in Bezug auf visuelle Qualität stetig an.

Reichenberg might oder maraschino Weiterentwicklung auf vielen Gebieten der Computergraphik wurde ein wesentliches Element für die Darstellung von Außenumgebungen stark vernachlässigt. Wolken können die visuelle Qualität einer Szene aufgrund ihrer komplexen Struktur, der Farbpracht sowie ihrer Entstehung und Bewegung äußerst stark beeinflussen. Da der Berechnungsaufwand zur Darstellung animierter und dynamisch berechneter Wolken beachtlich ist, war es mit handelsüblichen Grafikkarten nicht möglich diese darzustellen und gleichzeitig die Framerate hoch genug zu halten. Obwohl es viele Methoden gibt welche die Darstellung von Wolken für Außenumgebungen schnell genug ermöglichen, weisen diese alle einige Einschränkungen auf, sei es in der Bewegung, der Beleuchtung oder auch der Entstehung.

Diese Diplomarbeit untersucht Methoden die es ermöglichen plausibel wirkende Animationen von Wolken in Echtzeit mit Consumerhardware zu berechnen und anschließend die Ergebnisse dieser Berechnung zu nutzen um qualitativ hochwertige, dynamisch beleuchtete, volumetrische Wolken mit relativ hohen Frameraten darzustellen. Da visuell hochwertige Verfahren bereits für Nicht-Echtzeit Anwendungen existieren und diese so gut wie ausschliesslich von der CPU Gebrauch machen, was für die meisten Spiele nicht akzeptabel ist, da eben die CPU oftmals bereits mit voller Auslastung läuft, werden Methoden diskutiert um diese Algorithmen für die Berechnung mit der GPU anzupassen. Darüber hinaus werden auch einige neue Verbesserungen vorgestellt.

Die Diplomarbeit gibt zunächst einen Überblick über aktuelle Verfahren für Echtzeitanwendungen wie auch für Nicht-Echtzeitanwendungen. Danach werden Teile dieser Methoden für den Einsatz in interaktiven Umgebungen bei hohen Frameraten angepasst, während alle signifikanten Charakteristika der Nicht-Echtzeitmethoden wie die Entstehung von Wolken, die Bewegung und dynamische Beleuchtung erhalten bleiben.

Contents

1. Introduction

1.1. Real-Time Rendering

Real-time rendering has become one of the most important parts of computer graphics during the last years. Unlike non-real-time rendering algorithms, which can produce photorealistic results, real-time rendering requires nearly always a balance between performance and visual quality. Since the main purpose of this area is to give the user the feelings that he is inside the scene, it is necessary to keep the framerate high enough to produce smooth animations.

The term framerate describes the rate at which images are displayed. It's unit is defined as *frames per second* (fps), which will be used especially in Section 8 to analyze the performance of the algorithm proposed in this thesis. For most real-time applications, frame rates of 60 fps or more are recommended, especially if they include fast animations which need to be kept smooth. Nevertheless, it is not useful if the framerate exceeds the monitor refresh rate because this would imply that the user will never see some frames and the time to render these frames (especially under the consideration that every frame produces some amount of CPU overhead due to API and driver involvement) can be used more productively in most cases.

To achieve high framerates for rendering virtual environments, with visual quality being an important factor, the computation of images moved over the years from the CPU to graphics hardware, starting with the introduction of the 3dfx Voodoo graphics adapter about eleven years ago [1] (with the exception of professional workstations which used already some years earlier graphics accelerator hardware and some less successful graphics adapters from S3, Matrox, ATI and Hercules, which had driver problems that made them nearly useless for mainstream application). The 3dfx Voodoo chip already supported bilinear texture filtering (Figure 1.1), which significantly improved visual quality compared to real-time CPU computed images, which were not able to use this filtering algorithm because of performance reasons. Since then, the performance and features of GPUs (graphics processing units) continued to evolve with impressive speed. Graphics hardware has become mostly programmable now and it is possible to run complex programs for every single pixel which should be rendered, and it is also powerful enough to use multiple rendering passes for rendering at interactive frame rates (Figure 1.2). This allows not only nearly photorealistic results, it opens also new paths to perform general purpose computations which can be easily parallelized, which is especially the case for the algorithm proposed in this thesis.

Figure 1.1: Screen shot from Tomb Raider. The left image is rendered using the CPU (a Pentium 133 chip) without bilinear texture filtering at approximately 15 fps while the right image is rendered using a 3Dfx Voodoo graphics adapter with bilinear filtering enabled at 30 fps.

Figure 1.2: Nalu, created by NVIDIA to show the potential of Shader Model 3.0 graphics hardware. The real-time demo shows simulation of hair, advanced skin rendering, soft shadows and shafts of light from water surface. The image is rendered using 19 passes.

1.2. Importance of Cloud Rendering

„It is a nice day, isn't it?" „Yes, not a single cloud in the sky." – In virtual environments, unlike in the real world, the viewer will immediately notice the absence of clouds, since clear sky is not very common and also not very interesting to look at (as can be seen in Figure 1.3). In many computer games (which are the main application of real-time rendering) some kind of clouds can be seen. Until now, most of them are very simple and restricted in animation and lighting options – in fact, they are often just precomputed and mapped to a skybox. With increasing hardware capabilities and programmable GPUs, which are already widely available on consumer PCs, the expectations have also dramatically grown. Players don't want to see the same technologies (like static sky boxes) for years without any improvement, they want animated clouds which appear physically correct and with the same impressive formations and colors as in the real world.

This becomes even more important because for most games, cloud rendering is absolutely necessary. Therefore, a simple and efficient method is required that leads to nice visual results without nearly any artist work.

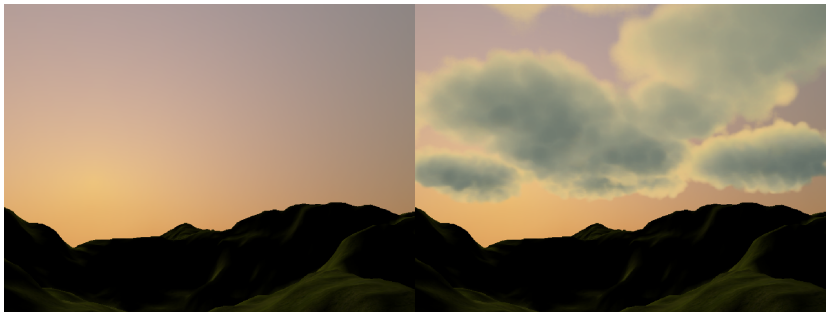


Figure 1.3: The left image shows a landscape at sunset with a clear sky. The right image shows the same scene with cloud rendering using the technique described in this thesis.

1.3. General Cloud Rendering Approach

The process of drawing clouds to the screen can be split into two main parts, cloud formation and rendering.

1.3.1. Cloud Formation/Animation

In the cloud formation and animation step, the (rough) shape of the cloud field in the sky is computed. This step could either be a pre-processing step (as it is the case in many games) or it could be dynamic (which is preferable). There are many different approaches to dynamically generate and animate a cloud field. Some of them will be discussed in Chapter 2 (related work) and one of them, based on cellular automaton, is used in a modified version for this thesis to compute the animation step entirely on the GPU.

Important for this step (which will shortly be called simulation step in the following chapters) is that the result needs not necessarily be a three-dimensional structure – in most literature, a two-dimensional image is generated and then used as heightmap for the second step. Examples for this are presented in Chapter 2.

1.3.2. Cloud Rendering

The cloud rendering step is used to draw the result of the cloud formation/animation (simulation) step to the screen. Usually, in this step also lighting computations are performed to illuminate the clouds. For this thesis, a part of the lighting computation (which computes shading for every single cloud particle) is not clearly within one of these two steps. In practice, it lies between them, but theoretically it could be done after each simulation step – then it would clearly belong to step one. Nevertheless it would also be possible to perform the shading computation before rendering the clouds – which is the most accurate (and also the most expensive) way. In this case, the shading step could be assigned to the cloud rendering step. In practically every other publication these steps are clearly split and the shading step is part of one of them (without implicitly mentioning them). As far as I know, the method proposed in this thesis, is the first which performs the shading step in such a flexible way, that accuracy vs. performance can be easily adapted (for example with a simple user-defined parameter).

Figure 1.4 shows a diagram with the steps for both – the method which is found in current publications and also the method presented in this paper. Of course, the rendering step has to be done for every single frame. Nevertheless, it is not necessary to call the simulation step more often than a few times per second, even for high-speed cloud movement animations it would be enough to perform the step for example every 1/10 second and then perform an interpolation between the last two steps for rendering. Since the simulation step is expensive for most methods, it is useful to perform it only if it is really necessary. This may be the main reason

why the two-step system is used within practically every publication about real-time cloud rendering. What is even more expensive (for most implementations) is the shading step, because it actually requires some sort of raycasting to determine how much light every particle of the cloud field receives. Since these computations are mostly implemented in the rendering step of current solutions, it decreases rendering speed dramatically. Therefore, I decided to design the method presented in this thesis a bit more flexible. The shading step lies now, as already mentioned, between the two classical steps. Depending on the performance of the GPU, it could be computed before each rendering step (which means, that shading is computed for every frame), but it can also be computed immediately after the simulation step, which may even be possible every few seconds.

This design can lead to a significant performance improvement concerning two simple facts: First, in practical scenarios, clouds move relative slowly, which means that it is not necessary to compute the simulation step very often – even thirty seconds or more may be enough, but it depends of course on the degree of modification per step. Second, which is also valid for most practical scenarios, the day-night cycle evolves at low speed – and shading recomputation is only necessary when the sun angle has altered for a few degrees since the last shading computation step.

A more detailed explanation will be given in the following chapters, this introduction to the steps of the algorithm is just necessary to understand the problems which exist on current solutions that are discussed in Chapter 2.

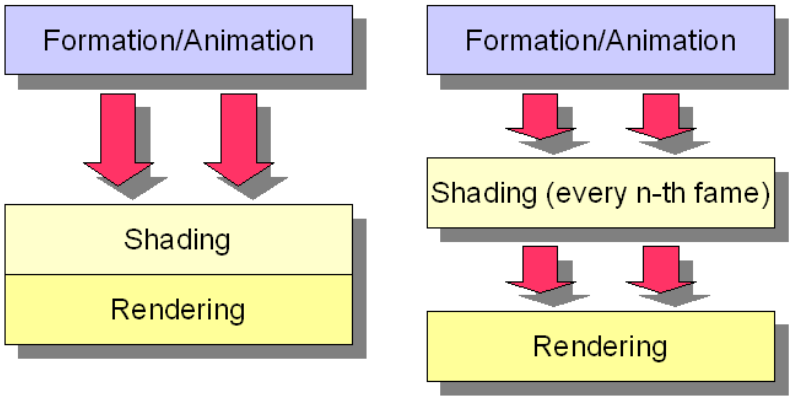


Figure 1.4: Two vs. three steps to draw clouds

1.4. Quality Criteria

Before it is possible to talk about advantages and disadvantages of previous work, it is necessary to define what is expected of a cloud animation and rendering system. There are a few quality criteria which are defined for a nearly optimal cloud renderer and which will be used to discuss previous work in Chapter 2 as well as the method proposed in this thesis.

1.4.1. Dynamic Simulation

- ✔ Realistic formation of clouds
- ✔ Cloud movement
  - ✔ advection by a single wind vector
  - ✔ advection by a more complex flow field
- ✔ Cloud extinction
- ✔ Nearly stable user-defined cloud density
- ✔ Control over cloud shapes; allow animators to define shapes which should be built during simulation

1.4.2. Dynamic Shading and Lighting

- ➢ Realistic lighting through scattering
  - ➢ single scattering, using one ray per cloud particle
  - ➢ multiple scattering, using multiple rays per cloud particle
- ➢ Advection by sun- and skycolor
- ➢ Local illumination phenomena (for example local lit particles near thunderbolts)

1.4.3. Performance Issues

- ➢ Possibility to perform cloud simulation in real-time
  - ➢ use GPU where it is possible and useful
- ➢ Possibility to perform cloud rendering in real-time
  - ➢ avoid memory transfers between main memory and graphics adapter.

1.5. Goals of this Work

Due to the increasing performance of GPUs, outdoor environments for games have become very popular over the last years. While algorithms for terrain rendering, water surfaces, shadows for large scale areas and vegetation rendering are constantly evolving and even if more and more games make extensive use of day-night-cycles (as can be seen in GTA San Andreas, Gothic I to III, etc.) there are still not many options to render beautiful skies with clouds.

The goal of this work was to implement a cloud rendering system which fits well to existing methods for sky rendering like the method proposed by Preetham et al.[2], which runs at interactive framerates. Also, the cloud rendering should match the quality criteria specified above with the limitation that single scattering was used because multiple scattering is still too expensive to be computed in real time. To achieve the goals, the cellular automaton method proposed by Nagel [3] and improved by Dobashi et al. [4][5] was adapted to be completely computed using the GPU and a new method to perform single scattering shading was developed.

1.6. Structure of this Document

This thesis is structured as follows: In Chapter 1, a short introduction will be given which explains why realistic cloud rendering is becoming more and more important. Then, in Chapter 2, related work, on which this thesis is based, will be shortly discussed. Chapter 3 introduces the algorithm proposed in this thesis before implementation details will then be discussed. In Chapter 4, a GPU implementation for volumetric cloud animation based on Dobashi's publications[4][5] will be presented. The proposed cellular automaton method will be improved by region maps, which are used to specify regions where clouds can appear. Dobashi already mentioned that ellipsoids could be used to specify such regions in a short section, but ellipsoids are not really useful for a hardware implementation, as discussed in Chapter 4. Chapter 5 shows a method to compute shading for each particle of the animation result in an efficient way that fits well to current GPUs. Then, in Chapter 6, a simple method to render the computed and shaded cloud particles is presented. Chapter 7 will introduce some performance improvements for particle rendering as well as a few extra features to increase the visual quality of the scene (like shafts of light, thunderbolts and cloud shadows on the ground), even if they are not directly part of this thesis. Chapter 8 will discuss the results of the proposed technique and it will be compared with methods introduced in Chapter 2. Finally, in Chapter 9, some ideas for improvements and future work will be shortly discussed.

2. Related Work

2.1. 3D Graphics Packages

Since real-time computer graphics shifted from CPU-based approaches to graphics hardware, 3D packages became ever more important as they are the bridge between the application and the graphics driver. They are necessary to keep hardware differences hidden from the programmer by providing a standardized application programming interface (API) that will work for every GPU and which includes methods required to program the hardware, for example drawing polygons, enable alpha blending or switching textures. This allows developers to concentrate on implementing the algorithms themselves instead of wasting time with porting the same feature for a wide range of graphics adapters.

Currently, the most popular graphics packages are OpenGL and Microsoft Direct3D, which is a part of DirectX (a complete library for game developers with various APIs for graphics, sound, input, etc.).

2.1.1. OpenGL

Unlike Direct3D, OpenGL is platform independent and implementations exist for most hardware platforms and operating systems. The API is preferable when the application should be portable or the operating system is not Microsoft Windows. In contrast to DirectX, OpenGL is based on extensions which are usually first implemented as vendor specific extensions. Therefore new features are usually available first for OpenGL, nevertheless the process of getting vendor extensions standardized through the Architecture Review Board (ARB) is very slow and developers often have to implement at least two different paths for newer features to make sure that they run at least on ATI and NVIDIA hardware.

2.1.2. Microsoft Direct3D

Direct3D is a part of DirectX and is available for Microsoft Windows only. Compared to OpenGL, DirectX was not built based on an extension system as it exists in OpenGL. This leads to the advantage that implementing features for

different hardware is more transparent, the API itself is responsible that a correct implementation runs on every GPU that fits the minimum system requirements. Of course, this comes also with a disadvantage – new features are in most cases first available for OpenGL because vendors deliver their OpenGL extensions before Microsoft is able to release a DirectX SDK update.

Since most games are released for Microsoft Windows and make use of DirectX, the demo application for this thesis is also implemented using the Direct3D API.

2.2. Graphics Pipeline

Modern (programmable) graphics hardware allows much more than just rendering a few triangles to the screen. It can be used for general purpose computations and makes it possible to perform the complete animation and shading step of cloud rendering on the GPU with nearly zero CPU overhead, as will be presented in Chapter 4. This section will give a short overview of the rendering pipeline and some details that should be considered for efficient GPU programming.

2.2.1. Overview

Since DirectX was used to implement the demo application related to this thesis, this chapter will mainly focus on DirectX; nevertheless current OpenGL implementations do not differ significantly.

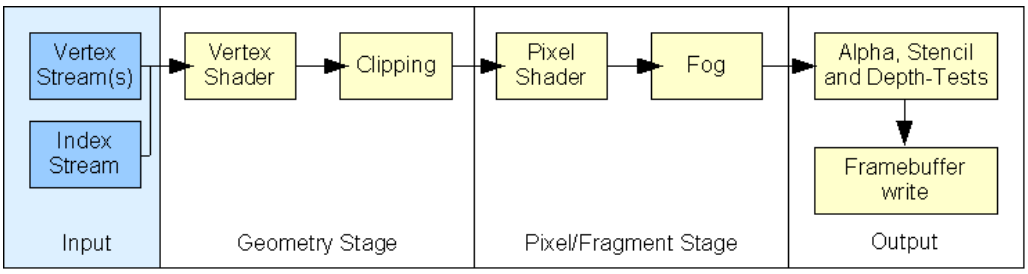


Figure 2.1: Direct3D pipeline without the outdated T&L engine and texture combiners.

As can be seen in figure Figure 2.1, the rendering pipeline consists of a few simple stages which are executed one after another. First, the user sets current vertex stream(s) and eventually an index stream if indexed polygons should be rendered.

The vertices are then transformed within the vertex shader, which takes exactly one vertex as input and writes exactly one vertex as output. On last generation hardware it is possible to create geometry dynamically using the so called *geometry shader*, but this feature is not necessary to implement the algorithm proposed in this thesis.

After vertices are transformed, clipping (and also backface culling) is performed. The remaining polygons are then rasterized (which occurs between the geometry stage and the pixel/fragment stage in Figure 2.1) and the pixel shader code is executed for every pixel. If fog is enabled, the pixel will be interpolated towards the fog color based on distance, but this is more and more replaced by user programmed fog using the pixel shader. Last, tests like alphatest, depthtest and stenciltest are performed to decide if the pixel should be written or not. Finally, if the tests do not fail, the pixel will be written (or possibly blended) to the framebuffer, which can be a texture as well on modern graphics hardware.

2.2.2. GPU Programming Issues

There are a few points that should be considered when developing software that uses the GPU. Most of them are well known from papers which are regularly released by GPU vendors like NVIDIA [6] and ATI, but I will mention the ones that are especially important for this thesis. Most of them are valid for NVIDIA as well as ATI, but since the demo application for the thesis is implemented on an NVIDIA GeForce 6800 Ultra, I will mainly focus on this vendor.

2.2.2.1. Draw Calls

Every draw call (for example DrawIndexedPrimitive, DrawPrimitive, etc.) will produce driver overhead that can quickly lead to CPU limitations in an application. Sim Dietrich[6] from NVIDIA Corporation suggests to keep the number of calls under 500 per frame for good performance. Since this includes all necessary passes to get the final image, especially shadow mapping passes, multiple passes for lighting, post processing, etc., 500 calls are not much and can be reached very quickly. Dobashi proposed in his paper [4] to render every single cloud particle with a separate draw call twice, one time for shading computation and one time for rendering. This leads to millions of calls, which can't even be performed in real-time by the fastest CPU. This was one of the main reasons why the proposed method was not suitable for real-time applications.

On modern hardware it is not a big deal to process a few polygons more which may even be not visible – it is nearly always useful to group polygons

that share the same materials (which means in most cases the same shaders and shader parameters) together and render them with a single draw call. Since performance for most games is not vertex shader limited (but very often CPU bound), grouping polygons together which can be rendered with a single draw call is one of the key aspects to achieve good performance.

2.2.2.2. Polygons per Draw Call

Don't perform draw calls with less than a few hundred polygons if it is avoidable. Calls with i.e. only two triangles will produce more driver overhead than they are worth. For this thesis, such calls are required for simulation and also for shading steps, but since they are not done very often (for example only every tenthousandth frame), it is acceptable. The reasons are nearly identical to that mentioned in Section 2.2.2.1. First, drawing less polygons per call will reduce the number of polygons that can be drawn since more calls would be necessary (which leads to more CPU overhead) and second, grouping polygons together if possible is always a wise decision since only a few applications are limited to vertex processing.

2.2.2.3. Only use Textures with Power-of-Two Dimensions

Most hardware is optimized to use textures with dimensions that are power-of-two, i.e. *32, 64, 128, 256,* etc. Some graphic cards do not even support textures with other dimensions and creation of such textures may fail. To support a wide range of hardware, keep in mind that only such textures should be used. For this thesis, that means especially that only volumes which can be organized on textures with power-of-two dimensions can (or rather should) be used. Information how volumes are organized on textures can be found in the Chapter 4, where a method is represented to perform the whole simulation step as proposed by Dobashi et al. [4] on the GPU.

2.2.3. General Purpose Computations using the GPU

General purpose computations are algorithms that usually need a high degree of freedom to perform them on a processing unit and that do mostly not fit very well to static environments like the Direct3D fixed function pipeline (even if some computations can also be performed using the FFP only, nevertheless they may be restricted in precision on older hardware – i.e. multiplications or additions using alpha blending).

Arithmetic algorithms basically consist of three simple components: input values, the computations itself and one or more output values.

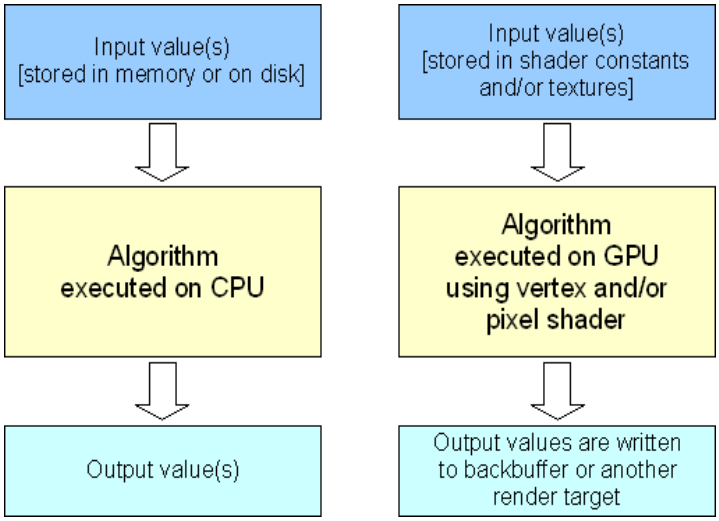


Figure 2.2: The left image shows how an arithmetic algorithm is typically structured for execution on the CPU while the right image shows how it is possible to execute such an algorithm using the GPU.

Figure 2.2 shows how general purpose computations can be performed on the GPU. Typically, input registers or textures are used to supply input data to the algorithm. Then, the algorithm itself is implemented as vertex- or pixel shader. In a typical scenario, a pixel shader is used because it is more flexible and powerful than vertex shaders on most hardware. The vertex shader itself is often just used to transform four (or six) vertices for a screen-aligned quad which is rendered to allow pixel shader execution for each pixel on the rendertarget. Of course it is also possible to use more than a single pass – i.e. the results of the first pass can be used as input for the following pass. Since modern hardware supports full precision (32 bit) in every pipeline stage, including all vertex- and pixel shader instructions, and nearly no restrictions in program flow (i.e. branches and iterations) on hardware that is capable of Shader Model 3.0, the GPU has become a powerful solver for arithmetic algorithms as well as other types of algorithms that can be partly executed using parallelism. This is also the main advantage compared to classical CPUs – graphics cards are, because of the rasterization process – designed for as much parallelism as possible. To give an example – the GeForce 8800 GTX has 128 shader processors which execute vertex- or pixel shader code in parallel for maximum performance. Since every output pixel is completely independent of other output pixels this is easily possible. Many computations that need such local results which are independent from other values may be much faster on the GPU than on the CPU.

There are already many publications available that discuss general purpose computations on the GPU [7] with a wide range of applications, reaching from global illumination computation over audio and signal processing [8] to scientific

computing [9] and bioinformatics applications [10]. A method which is quite similar to the volume simulation shader introduced in the next Chapter is Latta's paper *Building a million particle system* [11]. Textures are used there to store current particle states as well as to compute new states like positions, velocities, etc., which leads to massive performance benefits since it is no longer necessary to perform the complete particle animation on the CPU and then transfer the values back to the GPU (which is the most time consuming part). The simple idea behind that method is, that the particle states are stored on textures. To compute new states, the current states are used as input textures for the shader and a screen-aligned quad is used for rendering to execute the pixel shader for new state textures (which should have the same dimensions as the input textures) which are set as rendertargets.

Another really impressive application for general purpose computations on the GPU is NVIDIA's Gelato [12], which is a high-quality renderer that can use a cluster of NVIDIA GPUs as floating point processor units. The software is not based on the classical triangle rasterization which is typical for hardware-accelerated rendering, instead it uses the same methods as other renderers like povray, 3DS MAX, etc.

More very useful information on general computations using the GPU can be found on the GPGPU [7] website.

2.3. Cellular Automaton

Dobashi et al. [5] proposed in their paper *Animation of Clouds using Cellular Automaton* a simple, physically based model to perform animation of clouds. The cloud field was stored in a three-dimensional array (Figure 2.3) where each item stores three boolean values called *cld* (cloud particle), *hum* (humidity value) and *act* (activation bit).

- ***cld*** stands, as already mentioned, for cloud particle. If this boolean is set, a cloud particle exists on the current array position.
- ***hum*** stands for the humidity on the current position. According to the defined rules, clouds can only be formed where humidity is high enough – this simple fact is simulated using the humidity bit, which is used to describe the humidity distribution in the array.
- ***act*** stands for activation bit. This bit is initially set manually to start cloud formation on the specified position (of course, only if the humidity value is also set). Activation bits can be set at any time by the user to start new

cloud formations on specific array positions.

A few simple transition rules were defined which described how a cloud array at timestep t_i can be transformed to a new cloud array at timestep t_{i+1}.

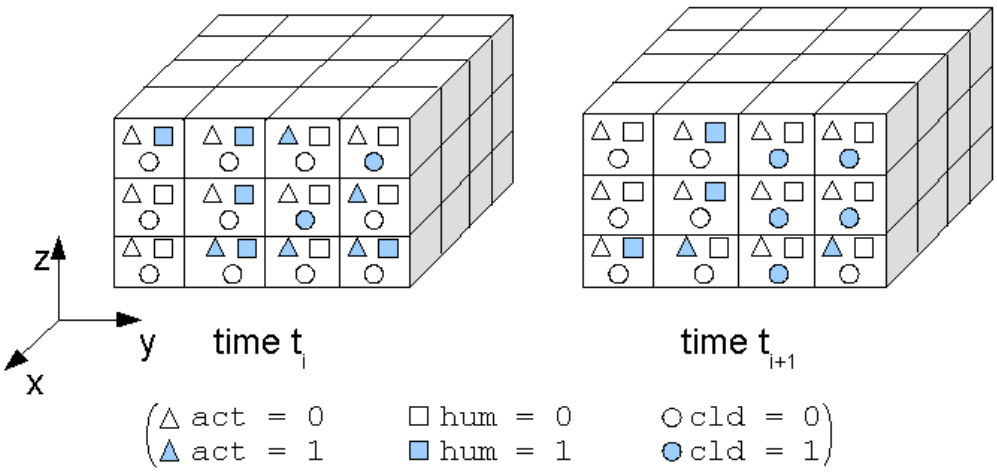


Figure 2.3: Three dimensional array, each item consisting of three boolean values.

2.3.1. Growth Simulation

$$hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \text{ and not } act(i,j,k,t_i) \quad \text{Equation 2.1}$$

$$cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \text{ or } act(i,j,k,t_i) \quad \text{Equation 2.2}$$

$$act(i,j,k,t_{i+1}) = \text{not } act(i,j,k,t_i) \text{ and } hum(i,j,k,t_i) \text{ and } f(i,j,k) \quad \text{Equation 2.3}$$

$$\begin{aligned} f(i,j,k) = {} & act(i+1,j,k,t_i) \text{ or } act(i,j+1,k,t_i) \text{ or } act(i,j,k+1,t_i) \text{ or} \\ & act(i-1,j,k,t_i) \text{ or } act(i,j-1,k,t_i) \text{ or } act(i,j,k-1,t_i) \text{ or} \\ & act(i-2,j,k,t_i) \text{ or } act(i+2,j,k,t_i) \text{ or } act(i,j-2,k,t_i) \text{ or} \\ & act(i,j+2,k,t_i) \text{ or } act(i,j,k-2,t_i) \end{aligned} \quad \text{Equation 2.4}$$

These rules work in the following way: humidity is only set in the next timestep if humidity already exists in the current step and no activation bit is set. The cloud particle bit is only set if there is already a cloud particle on the position or if the activation bit for the position is set – which means, that the transition from vapor to cloud is ready. Activation bits are a bit more complex, because they are used to define the general shape of clouds on a physically based model. In this model, clouds expand more in higher areas than in lower ones, which is defined by the fact, that there is no *act(i,j,k+2)* in the formula that defines *f(i,j,k)*. An activation bit is only set if there is currently no activation bit on the position, if

humidity exists and if *f(i,j,k)* evaluates to true, which means that at least one activation bit must exist in the neighborhood of the current position. This makes sure, that it looks like clouds are growing.

As can be seen, according to the quality criteria in Section 1.4, there are a few problems with the defined transition rules. First, whenever a cld bit is set, it can never disappear. Assuming that a uniform distributed humidity value is used over the whole array, the clouds would cover the full sky after a few simulation steps and they will never disappear. Second, humidity regeneration is not included in this model. Whenever a humidity bit is „used", it disappears and will be set to *false* for the rest of the simulation. Third, advection by wind is also not defined, which makes it very unrealistic. Besides, the full simulation step is performed on the CPU, which requires to transfer memory after each simulation step to the graphics card. Even if this would not be necessary, the CPU is in many computer games the bottleneck, which makes it not practical to perform such computations there.

Some of the mentioned problems were solved in Dobashi's paper *A Simple, Efficient Method for Realistic Animation of Clouds* [4]. Dobashi et al. improved Nagel's method [3] in four points:

- Extinction of clouds
- Advection by wind
- Increased simulation speed
- Control of cloud motion

The already known transition rules were used for growth simulation without any modification. There were just a few rules added for cloud extinction and advection by wind.

2.3.2. Cloud Extinction

The extinction rules are based on probability values which can be defined for each item in the three-dimensional array.

$$cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \text{ and } IS(rnd > p_{ext}(i,j,k,t_i)) \quad \text{Equation 2.5}$$

$$hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \text{ or } IS(rnd < p_{hum}(i,j,k,t_i)) \quad \text{Equation 2.6}$$

$$act(i,j,k,t_{i+1}) = act(i,j,k,t_i) \text{ or } IS(rnd < p_{act}(i,j,k,t_i)) \quad \text{Equation 2.7}$$

These rules are used to produce a simulation that could theoretically become stable and so it is possible to use it for continuous cloud animation. Existing cloud

particles are removed from the simulation if the p_{ext} value of the position is greater than a randomly generated value. This makes it possible to define regions where extinction of clouds may frequently occur while other regions can be defined where extinction rarely occurs.

Humidity is only set in step *i+1* if humidity is already *true* in step *t* or a random generated value is less than p_{hum}. This avoids the above mentioned problem, that consumed humidity values are never set again to *true*.

New activation bits are added to the simulation to allow formation of new clouds based on the probability value p_{act}.

2.3.3. Advection by Wind

Dobashi [4] introduced a really simple method to simulate advection by wind. The cloud array is just moved, so the wind always affects the complete volume, local influences are not possible.

$$hum(i,j,k,t_{i+1}) = \begin{cases} hum(i-v(z_k),j,k,t_i), & i-v(z_k) > 0 \\ 0, & otherwise \end{cases} \quad \text{Equation 2.8}$$

$$cld(i,j,k,t_{i+1}) = \begin{cases} cld(i-v(z_k),j,k,t_i), & i-v(z_k) > 0 \\ 0, & otherwise \end{cases} \quad \text{Equation 2.9}$$

$$act(i,j,k,t_{i+1}) = \begin{cases} act(i-v(z_k),j,k,t_i), & i-v(z_k) > 0 \\ 0, & otherwise \end{cases} \quad \text{Equation 2.10}$$

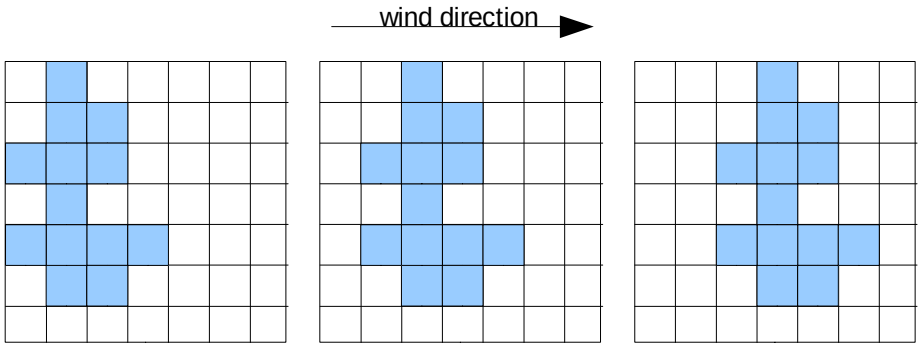


Figure 2.4: Advection by wind on a single slice of the volume. The left image shows timestep t_{i-2}, the center image shows t_{i-1} and the right image shows timestep t_i.

2.3.4. Efficient Cloud Field Simulation using the CPU

As already mentioned, the complete simulation step (including growth simulation, cloud extinction and advection by wind) is computed by the CPU. Dobashi et al. [4] proposed a method to accelerate computation and at the same time reduce memory consumption. The basic idea is, that a 32 bit CPU could perform 32 boolean operations in a single step. Since all three components of a cloud array item *(cld, act, hum)* are boolean values, they can be evaluated at the same time. This raises the necessity to use only power-of-two dimensions for the cloud field, to make sure that not a single byte of the integers is wasted, which is not really a problem. The method presented in this thesis imposes the same restriction.

2.3.5. Rendering of the Simulation Results

Even if the CPU-based simulation of the cloud field can be done in real time (using the bitfield manipulation approach), the rendering algorithm proposed by Dobashi et al. [4] is very slow. Generally, it works in three steps.

2.3.5.1. Smoothing of the Cloud Field

In a first step, it is necessary to smooth the simulation result over space as well as over time. This is necessary for two reasons – first, the distribution of cloud particles is discrete, so there are no density values different from 0.0 or 1.0; second – what is even more important – it is necessary to perform some sort of interpolation over time, i.e. over timestep t_i and t_{i+1}.

Another three-dimensional array of float values is used, so the memory consumption advantage using single bits to represent cld, hum and act is lost. In addition, more instructions are necessary to read a single *cld* value because of the encapsulation into integers values. So even if the simulation step may be fast on the CPU, it is very difficult to get the smoothing step running at high framerates.

Smoothing is performed by computing average values using neighboring cells. Dobashi et al. [4] proposed the following formula to do this:

$$q(i,j,k,t_i)=\frac{1}{(2i_0+1)(2j_0+1)(2k_0+1)(2t_0+1)}$$
$$\sum_{i'=-i_0}^{i_0}\sum_{j'=-j_0}^{j_0}\sum_{k'=-k_0}^{k_0}\sum_{t'=-t_0}^{t_0} w(i',j',k',t')\,cld(i+i',j+j',k+k',t_i+t')$$

Equation 2.11: cloud field smoothing

For this equation, w is a weighting function ensuring that cells near (i,j,k,t_i) have a higher priority than cells which are further away.

2.3.5.2. Computation of Color for every Cell

Dobashi et al. used billboards to render the clouds. Simply speaking, every cell where the cloud density is different from 0.0, is assigned a precomputed billboard texture. For each billboard, it is necessary to compute the color which should be used in the final pass.

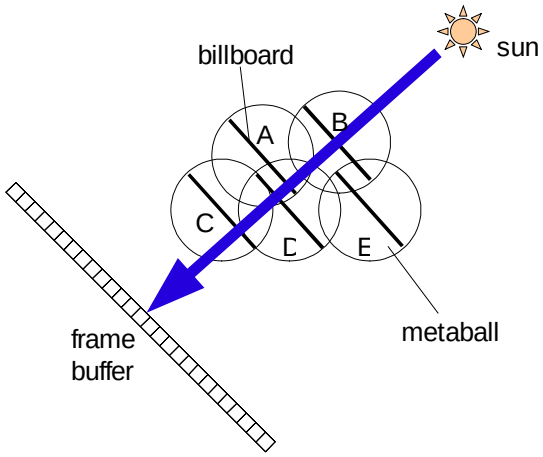


Figure 2.5: Each billboard is rendered from the sun position in an ascending order and the framebuffer is read after each draw call to determine the attenuation ratio.

The image is first rendered as seen from the sun. The frame buffer is initialized as 1.0 (for all three components `r,g,b`). Now, the billboards are placed at the center of each cell and oriented towards the sun. The attenuation ratio for each billboard (they are surrounded by so called metaballs in Dobashi et al. [4]) is computed by first sorting them in ascending order based on their distance from the sun. Then, each of them is rendered to the framebuffer with alpha blending enabled. The pixel values in the buffer are multiplied by the attenuation ratios stored within the billboard textures. For example, the attenuation ratio between metaball C and the sun in Figure 2.5 is obtained by multiplying the attenuation ratio of metaballs A, B and D, because each one of them absorbs some amount of light emitted by the sun. After rendering A, B and D to the framebuffer, the pixel value in the framebuffer on the position where the projected center of metaball C is located is read back to obtain the attenuation ratio. The color of the metaball can be evaluated by simply multiplying the read attenuation ratio with the sun color. After all billboards are rendered, the

framebuffer holds a lightmap which can be used to project shadows on the ground or even to render shafts of light.

2.3.5.3. Render Clouds to Screen

Now since the color for every billboard is known, rendering of the clouds is straightforward. First, all scene objects are rendered. Next, the billboards are oriented towards the viewer and rendered in back-to-front order using the previously computed colors with alpha blending enabled.

2.3.6. Quality Analysis

2.3.6.1. Dynamic Simulation

Formation of clouds and extinction are completely solved and seem to be physically plausible. Advection by wind is implemented, but local turbulences are not considered, though it would be theoretically possible to do them by manipulating ellipsoids with local wind vectors. A quite similar approach is introduced in Section 4.3.3. Nevertheless, Dobashi et al. [4] proposed no method to control the cloud density and perform plausible animations without heavy user interaction.

2.3.6.2. Dynamic Shading and Lighting

For cloud shading, a single-scattering approach is used – which means, that light only propagates into one direction (along the direction vector of sun light). In theory, multiple scattering of light between particles should be considered for clouds, but in practice a simple ambient value works fine instead. The results are impressive and the shading method proposed in this thesis is basically a real-time implementation of Dobashi's shading algorithm.

2.3.6.3. Performance Issues

The simulation step is done completely using the CPU, which may not be such a big drawback (according to speed) because it can be computed very efficiently using the proposed bitfield manipulation functions. Nevertheless, a CPU implementation requires to transfer the simulation step results to the graphics card, consuming bandwidth and it may even be necessary to synchronize graphics card and CPU because a buffer within graphics card memory has to be locked before data can be written to it [13].

The shading step is implemented very inefficiently and does not fit well to nowadays graphics hardware. Sim Dietrich from NVIDIA Corporation recommends in his *Modern Graphics Engine Design* slides [6] to keep the number of draw calls under 500 for good frame rates. Since the method proposed by Dobashi et al. requires to render each billboard separate, followed by a transfer of the framebuffer from graphics memory into main memory to read the attenuation ratio pixel value this cannot be done in real-time. According to the results section of Dobashi's paper [4], rendering of an image (including the shading step) took 10-30 seconds on an (as of today) outdated graphics card. Because of the high number of necessary draw calls, it is not much faster on current hardware.

The rendering step suffers the same problem as the simulation step – the high number of draw calls. Even if it is not necessary for this step to copy the framebuffer after each particle is rendered, the number of draw calls (for a 256x32x256 volume, the worst-case scenario is more than 2 million calls) will produce so much driver overhead, that it is impossible to render it with interactive frame rates.

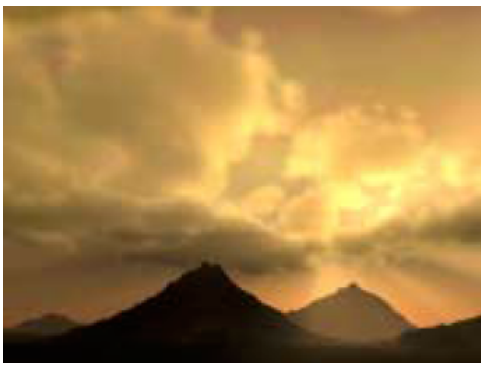


Figure 2.6: Rendered cloud field from Dobashi's paper[4]

2.4. Cloud Rendering using Impostors

Harris proposed in *Real-Time Cloud Rendering* [14] a method for fast cloud rendering using impostors. This method works well for flight simulators and games, but it has some significant drawbacks. First, animation of clouds is not considered and second (what is even a bigger problem), the shading is done in a precomputation step to allow multiple scattering. The method allows to render beautiful clouds at very high framerates, by using impostors, which replace

hundreds or thousands of particles by an approximated image version available for various viewing angles. However, the restrictions are problematic.

Generally, cloud rendering is performed in two steps, similar as in Dobashi's paper [4]. In the first step, shading is computed and in a second step, the shading step results are used to render the clouds.

2.4.1. Multiple Forward Scattering

The algorithm presented by Dobashi et al. [4] used, as already mentioned above, only single scattering for shading computation. According to Max's paper *Optical Models for Direct Volume Rendering* [15], single scattering only leads to a useful approximation if the albedo or the density is low. For clouds, this is usually not the case, which makes it necessary to find efficient scattering approximations to implement realistic shading, otherwise the clouds (using single scattering and realistic values for extinction and albedo) will appear very dark on the side facing away from the sun, because the light is scattered only in one direction and some amount of light is absorbed whenever a particle is hit.

True multiple scattering is very expensive (since it solves light propagation in every direction) in terms of computation time and is also not necessary even for most scientific visualization applications [15].

To find a useful balance between accuracy and computation time, Harris [14] introduced the so-called multiple forward scattering. As shown by Nishita et al. in their paper *Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light* [16], the contribution of most of the paths used for multiple scattering is insignificant. Scattering illumination is dominated by the first and second orders. This is the reason why they simulated only up to the fourth order. They also reduced the directions by concentrating on sub-spaces of a high contribution. Harris simplified this further and approximated multiple scattering only in the direction of the light – which led to the denotation *multiple forward scattering*.

The following equations, which describe the scattering method, are discussed in more detail in Harris' paper [14].

$$I(P,w) = I_0(w)\, e^{-\int_0^{Dp} \tau(t)\,dt} + \int_0^{Dp} g(s,w)\, e^{-\int_s^{Dp} \tau(t)\,dt}\, ds \qquad \text{Equation 2.12}$$

$$g(x,w)=\int_{4\Pi} r(x,w,w')\,I(x,w')\,dw' \qquad \text{Equation 2.13}$$

P	*Position vector*
w	*Light direction*
I(P,w)	*Sum of direct light from direction w that is not absorbed by other particles and light scattered to P from other particles.*
$I_0(w)$	*Intensity of light in direction w outside the cloud*
$\tau(t)$	*Extinction coefficient of cloud at depth t*
Dp	*Depth of position P within the cloud regarding to light direction*
g(x,w)	*Light from all directions w' scattered into direction w at the point x.*
r(x,w,w')	*Bi-directional scattering distribution function (BSDF)*

Even if multiple scattering will not be used for the method presented in this thesis, the formulas for multiple forward scattering are shown, because it should be possible to modify the shading algorithm introduced in Chapter 5 to perform this type of scattering in a GPU friendly manner. For now, single scattering only is used to increase performance as much as possible, since – as already mentioned in the introduction section – the main application of real-time rendering are computer games, where performance is still more important than visual accuracy. To avoid very dark clouds, an ambient term will be used instead which allows to avoid this effect and produce plausible results.

2.4.2. Quality Analysis

2.4.2.1. Dynamic Simulation

Animation of clouds is not considered in Harris' paper[14] and it would be really hard to implement because of the pre-processing that is necessary. So, it works fine for static environments and without day-/night cycle, but it is not very useful for dynamic worlds where cloud formation, motion and real-time shading are required.

2.4.2.2. Dynamic Shading and Lighting

Shading is well approximated using multiple forward scattering and it yields very impressive results, as shown in Figure 2.5. Even if it is not a true

multiple scattering approach where sampling is done in all directions, the multiple forward scattering approach works really well for clouds and it could be used to render high-quality images.

2.4.2.3. Performance Issues

The method presented by Harris [14] was only usable for static environments, because the scattering computations were performed on the CPU. Furthermore, simulation of clouds was not considered.

Figure 2.7: Top view: clouds rendered with Harris' impostor technique [14] using single scattering (as can be seen, the side facing away from the sun appears unnaturally dark). Bottom view: the same scene rendered using multiple forward scattering, which leads to much more realistic results. Note that the upper cloud layer is just part of a skybox (or a similar technique) and not produced by the method presented in the paper.

2.5. Cloud Dynamics Computation using Fluid Flow Equations

In [17], Harris introduced a method to perform computation of cloud dynamics on the graphics hardware using a model based on fluid flow equations, which is physically more accurate than Dobashi's cellular automaton technique [4] but at the same time very time consuming, even for small volumes. Apart from that, the cellular automaton technique produces realistic looking results and does not need bounding conditions (which usually further reduce the usable size of the volume). Finally, the cellular automaton method could also be implemented using boolean operations on latest graphics hardware, which means that 32 voxels could be processed at only a few GPU cycles. Nevertheless, the general idea of encoding volumetric data into a 2D texture for GPU processing introduced by Harris in that work is also used for this thesis to implement Dobashi's cellular automaton method [4].

Harris also proposed a method for real-time cloud shading. The idea is, to convert the volumetric data (which is obtained from a simulation step based on fluid equations) encoded in a 2D texture into a 3D texture for shading computation. The volume is enclosed in a so-called light volume which is oriented in light direction. Slices are then rendered for the light volume. Every particle hit absorbs some amount of light, which is propagated from the slice next to the light source to the slice farthest away. Figure 2.8 shows the light vector aligned bounding box used to enclose the cloud volume (which is already described by Dobashi et al. [4]). Instead of single particles, Harris renders complete slices by performing texture lookups in the 3D cloud texture.

This shading method has two serious drawbacks: First, it is necessary to convert the cloud volume which is stored as a series of slices on a 2D texture, to a 3D texture. Since most graphics cards do not support 3D textures as render targets, it is necessary to lock the volume texture and copy the data using the CPU, which means that a huge amount of data needs to be transferred from system memory to graphics memory. Second, the method wastes many pixels of the light volume slices instead of using them for cloud shading, as can be seen in Figure 2.8. The wasted areas are marked grey. Depending on the light vector, a large light volume may be necessary to compensate this.

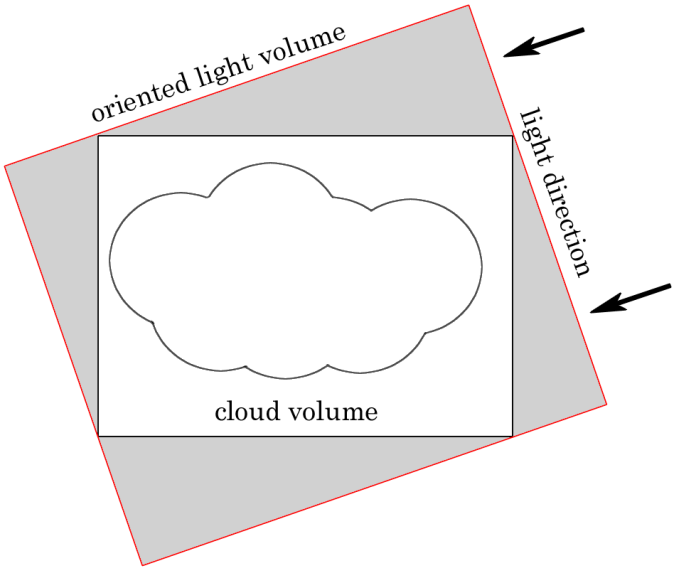


Figure 2.8: Harris proposed to enclose the cloud volume into a bounding box as a light volume that is oriented in light direction to compute shading. Slices are positioned in this light volume and cloud particle densities are read from the previously converted 3D cloud volume to compute attenuation.

2.5.1. Quality Analysis

2.5.1.1. Dynamic Simulation

Simulation is performed on graphics hardware using an approach based on fluid flow equations. This is physically more accurate than the cellular automaton method proposed by Dobashi et al. [4].

2.5.1.2. Dynamic Shading and Lighting

The method supports dynamically scattering computations. As can be seen in Figure 2.8, the quality of shading depends highly on the orientation of the light direction vector.

2.5.1.3. Performance Issues

Computation of the simulation step using fluid flow equations is very expensive. Even for small volumes, the computations take a long time and need to be split over multiple frames. The shading algorithm suffers the problem that it it necessary to copy the cloud volume (which is stored on a 2D texture) to a 3D texture for shading computation.

2.6. Perlin Noise based Methods

Most current work about cloud rendering is based on Perlin noise maps, which were introduced by Ken Perlin [18]. These maps can be generated very fast (even GPU implementations exists) and plausible results (to the user) can be produced using them, even if they are not based on a physically model.

2.6.1. Cloud Formation and Animation

Kim Pallister describes in *Generating Procedural Clouds in Real Time on 3D Hardware* [19] how Perlin noise maps can be used to create a cloud texture which is then mapped to a curved surface on the sky and rendered with alpha blending enabled.

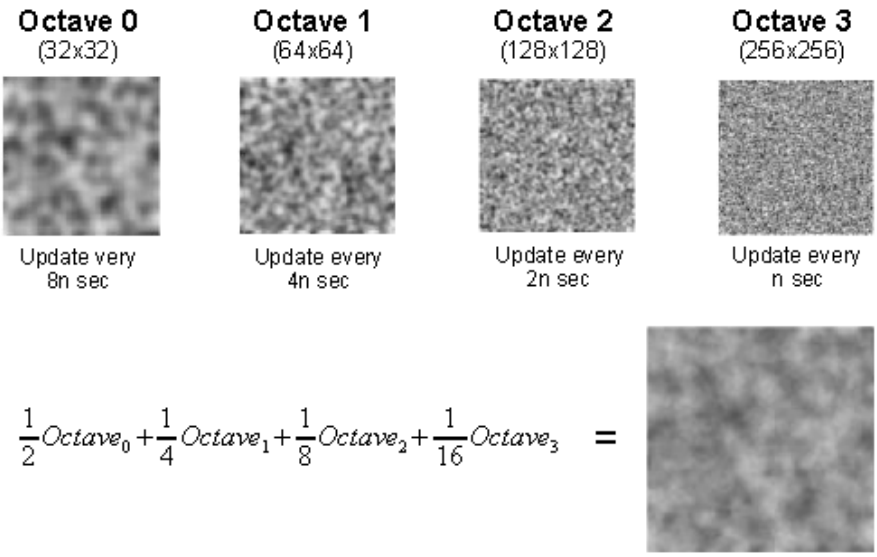


Figure 2.9: Perlin noise maps with different frequency are added to create a cloud texture [19].

As can be seen in Figure 2.9, four Perlin noise maps with different octaves are combined in an additive way to produce a more plausible cloud map. This allows only some sort of simple animation. $Octave_0$ specifies the rough shape of the clouds, and because the general cloud shape changes very slowly, it is only updated every 8n steps. The other octaves with higher frequency are used to add more detail to the clouds. The higher the frequency, the higher the rate of change, because small details change very fast while the base shape remains nearly constant over a long time.

Since cloud textures generated in this way practically always cover the whole sky (which is mostly an unwanted effect) it is necessary to modify them. Pallister [19] proposed to clamp against a constant value to create isolated clouds, as can be seen in Figure 2.10.

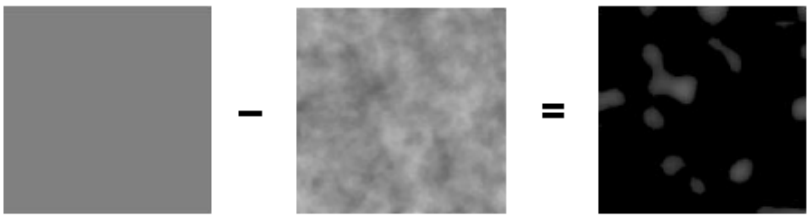


Figure 2.10: Map clamped using a constant to create isolated clouds [19].

Figure 2.10 shows how isolated clouds can be generated using Perlin noise. Nevertheless, these isolated clouds do not look very natural and it looks like they suffer from too less detail, which means they look flat.

Dubé presented in the *Game Programming Gems 5* article *Realistic Cloud Rendering on Modern GPUs* [20] an improvement to the algorithm proposed by Pallister [19], which solves the problem that occurs after subtraction.

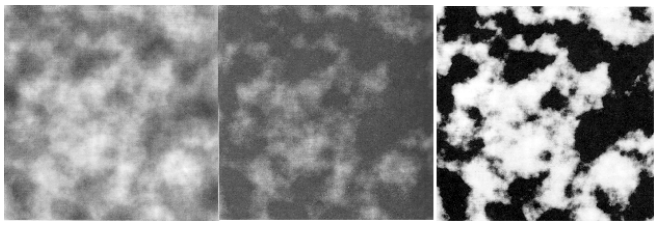


Figure 2.11: Exponentiation is performed after subtraction to increase quality [20]. The left image is the map which is computed by adding the four noise maps, the center image shows the same map after subtraction and the right image shows the center image after exponentiation.

As can be seen in Figure 2.11, exponentiation makes sure that the final map appears more cloud-like. This modification is very simple and can be done directly within the pixel shader.

2.6.2. Rendering Clouds using the Generated Map

Pallister [19] used a very simple approach to render the clouds. In fact, he just enabled alpha blending and projected the generated texture map to a plane or a curved surface above the viewing point. This method does not allow any cloud shading and the final image does not look very impressive as can be seen in Figure 2.12.

Figure 2.12: Final image from Pallister's paper [19].

Even if shading were implemented using this method, it would still suffer from visible filtering artifacts, because texture resolution (the proposed resolution is 256x256 pixels) is too low. Experiments lead to the insight that even a resolution of 2048x2048 pixels is not enough to cover the whole sky.

Dubé [20] chose a more expensive rendering approach which is performed completely on the GPU. It is based on casting a ray through the whole cloud for each screen pixel. For this solution, the heightmap is interpreted as a volume as shown in Figure 2.13.

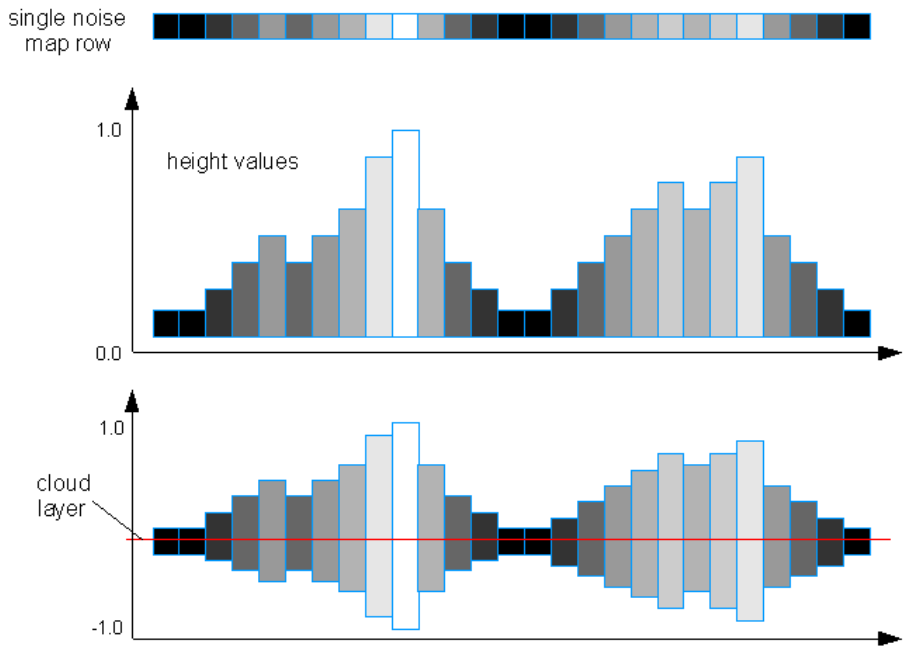


Figure 2.13: The top image shows a single pixel row of the cloud map. The center image shows the same row interpreted as height values (as is well known from terrain rendering) and the bottom image shows the same interpretation but from each value 0.5 is subtracted and then multiplied with 2.0 to get the values into a [-1.0, +1.0] range where 0.0 represents the cloud layer.

Since raycasting is used on a per-pixel basis on the rendertarget (and not on the texture itself) the sampling is much better and the final image does not suffer from filtering artifacts even if the input texture dimensions are only 512x512 pixels. The raycasting approach also allows implementing single scattering (even multiple-scattering is possible, but not practical) and leads to great results as shown in Figure 2.14.

Figure 2.14: Final image from Dubé's article [20].

2.6.3. Quality Analysis

2.6.3.1. Dynamic Simulation

Simulation when using noise maps is primarily achieved by rotating the texture and changing the degree of cloud cover by modifying the constant value for subtraction. Especially the second one does not look very realistic for fast simulation speeds, nevertheless it looks plausible for slow speed, even if no physically based model is used.

The shape of the clouds can't be specified by the artist. Even if a rough control may be possible, the method is not nearly as flexible as the simulation step presented by Dobashi et al. [4].

2.6.3.2. Dynamic Shading and Lighting

The method presented by Pallister [19] does not include any shading computations at all, therefore this algorithm is impractical for today's requirements. Dubé [20] solved this problem and implemented a shading algorithm that uses single scattering with some artistic improvements which are not physically based (of course this is always necessary for a single scattering approach to avoid clouds that look too dark). Also note, that the form of the clouds is restricted as shown in Figure 2.13, because it is always completely identical above and below the cloud layer.

2.6.3.3. Performance Issues

The simulation step can be done in real time at high framerates, since Perlin noise maps can be generated entirely on the GPU. The rendering step as proposed by Dubé [20] is very slow and will not scale well to current game environments. Even if the complete work were done on the GPU, the raycasting approach is still expensive and iterations (which are necessary to step through the volume) are not really fast on today's graphics hardware.

Experiments with the proposed algorithm led to the conclusion that the performance highly depends on screen resolution. On a GeForce 6800 Ultra graphics card, the rendering algorithm runs at nearly 60 frames per second on a screen resolution of 640x480. It drops down to 30-40 frames when using 800x600, which is still not enough for todays games. According to the fact, that most games have to render more than just a few clouds, the proposed method may not be usable in the next years.

It will also require hardware that supports at least Shader Model 3.0, which is already becoming more and more mainstream, but until now, Shader Model 2.0 hardware needs to be supported as well in most games. The method proposed in this thesis will also use Shader Model 3.0 hardware, but it would be possible to do a Shader Model 2.0 hardware implementation by splitting the work of the simulation step into several passes.

3. Algorithm Overview

The proposed algorithm is mainly based on Dobashi's paper *A Simple, Efficient Method for Realistic Animation of Clouds* [4]. The technique described in this paper is not suited for interactive environments, so methods will be proposed to perform nearly all computations on the GPU and as a result allow cloud simulation and rendering at interactive frame rates.

The technique can be split into three parts, which are: The Simulation step for computing volumetric cloud data, the shading step for computing shading information for the previously generated cloud field, and the rendering step, which draws dynamically generated and shaded clouds to the screen. A short overview can be seen in Figure 3.1.

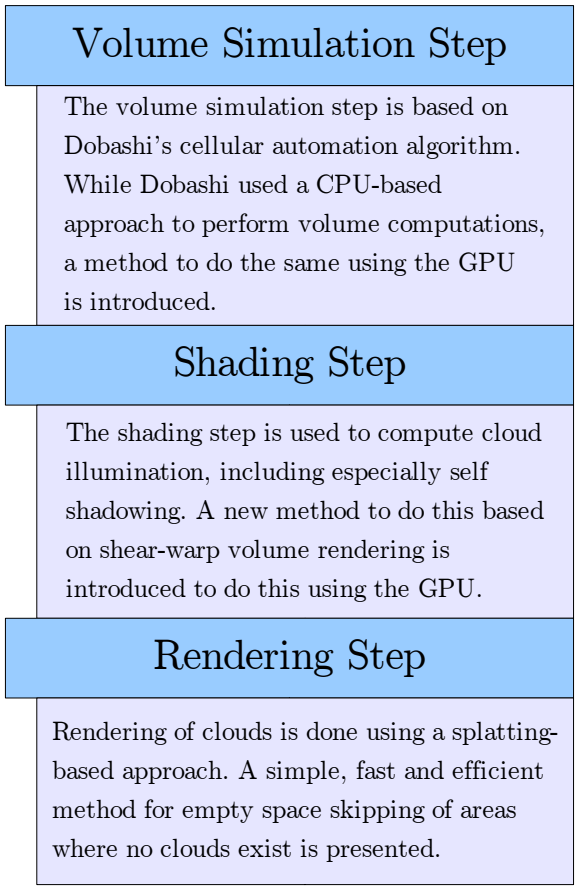


Figure 3.1: The technique proposed in this thesis can be split into three main steps (simulation step, shading step, rendering step).

In the following subsections each of the three steps will be shortly explained and the new contributions will be explicitly listed. The following steps will explain the three steps in detail.

3.1. Simulation Step

For cloud volume simulation Dobashi's cellular automaton algorithm [4][5] is used. This algorithm implements only a few simple equations, called transition rules, which specify how a new volume for timestep t_{i+1} can be computed using an existing volume which represents timestep t_i as input.

Since the computations depend only on the previous timestep, the volume simulation can be easily parallelized and therefore it is well suited for a GPU-based implementation, which will be presented in Section 4.

The following simulation step features are new:

- Volume computation is completely performed on the GPU. Dobashi et al. [4] used a CPU-based approach. Another method to perform cloud field computations on the GPU is already proposed by Harris [17], but a GPU based method for Dobashi's cellular automaton does not exist.
- An efficient method to control cloud formation and movement using a *region volume* is introduced.
- Advection by wind is improved using multiple vectors to control cloud motion.
- New parameters to allow easy control of average cloud size, density of the cloud field and lifetime of clouds is presented.
- A simple method to roughly reconstruct clouds which have already moved out of the volume during advection by wind is introduced. This is especially interesting for fast movements when direction changes happen (for example as it is the case for holding patterns in flight simulations).

3.2. Shading Step

For cloud shading, a single scattering approach is used. Shading computations for the complete cloud volume are performed using a new technique, which is called inverse shear-warp, since it is based on the shear warp volume rendering algorithm [21]. Volume slices are copied shifted by the light direction vector within the pixel

shader to propagate attenuation through the volume.

The following shading step features are new:

- A new shading technique is proposed that is based on the shear-warp rendering method.
  - Maximum precision for shading computations. Shading map pixels are not wasted, which is a problem in other approaches. For every voxel in the volume, a corresponding shading value is computed.
  - Entirely on the GPU. CPU overhead is reduced to set shader parameters and compute the offset to shift slices.
  - Easy computation split over multiple frames.

3.3. Rendering Step

For cloud rendering, a splatting-based approach is used. Point sprites are chosen as primitive type since they can be rendered very fast because only one vertex needs to be transformed per primitive. To keep the number of draw calls low, point sprites are grouped in regular grids (for example storing 32x32 particles). Whether a particle needs to be rendered or not is decided within the vertex shader, which means that it is not necessary to dynamically adjust the vertex stream for each situation; the same stream, which is created in a preprocessing step, can be used to render the whole cloud field.

The following rendering step features are new:

- Only one vertex stream is necessary to render the whole volume. The vertex shader is used to decide if individual point sprites need to be rendered or not.
- A fast and efficient method to perform empty space skipping. This approach allows skipping regions where no clouds exist.

4. Animating Volumetric Clouds using the GPU

This is the first chapter that discusses details of the method proposed within this thesis. As already mentioned in Chapter 1, cloud rendering is always performed in 3 steps. The first of these steps is the simulation step, on which I will focus in this chapter.

4.1. Cellular Automaton using the Render-to-Texture Feature

For simulation of cloud formation and animation, the cellular automaton given by the equations introduced by Dobashi et al. [4] will be used. This approach is modified to be performed entirely on the GPU.

4.1.1. Organizing Volumetric Data on a 2D Texture

Since three-dimensional textures can't be used as render targets in DirectX 9 (which is the target API for the demo application related to this thesis), it is necessary to perform the three-dimensional volume simulation step on 2D textures. The solution is quite simple, a 2D texture with four channels (red, green, blue, alpha) is used to store the whole volume. A similar approach is described in [17]. This texture is split into slices. Since the cellular automaton algorithm only needs results from the previous step and is fully parallelizable, it is predestined to be performed on the GPU. The main drawback when using a 2D texture is that it is only possible to use bilinear filtering instead of trilinear filtering which could be used for volumetric textures. However in the case of the cellular automaton algorithm, accuracy is important and interpolated (and thereby filtered) values are not desired.

Figure 4.1 shows a sample configuration for a 2D texture with 2048x1024 pixels which stores 32 volume slices – each of them sized 256x256 pixels. This leads to approximately 2 million cells and consumes about 8 MB of graphics memory (because four components – each of them 8 bits – are used). Of course it is possible to choose any other number of slices and texture width/height (as long as the hardware supports it – the GeForce 6800 Ultra supports textures with a maximum

width/height of 4096x4096 pixels). Keep in mind that power-of-two dimensions should be used wherever it is possible. Furthermore, it is not practical to use more

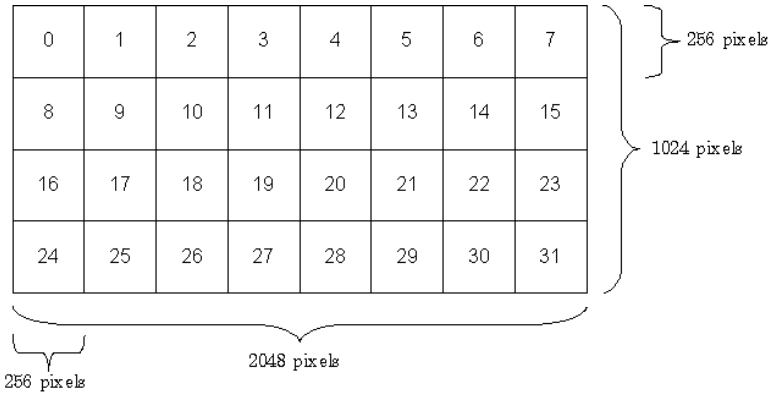


Figure 4.1: Sample configuration of a 2D texture used to store 32 slices to form a volume.

than 32 slices. The clouds may become unnaturally high and memory consumption should be considered for such a large texture (which is required two times, one for timestep t_i and one for t_{i+1}).

For the remainder of this thesis, slice 0 is defined to be the slice which has the smallest distance to the ground and the slice with the highest index (which is slice 31 in Figure 4.1) has the greatest distance, which makes it the highest layer.

4.1.2. Using Three Texture Components to store cld, act and hum Values

As already mentioned, a texture format with four components is chosen to store the volume(s). There are two reasons for this – first, it will be possible to store all necessary booleans *(cld, act, hum)* as described in Dobashi's paper [4] and second, it is mostly supported on current hardware to be used as a render target. The alpha-channel will not be used for the moment.

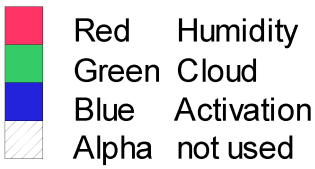


Figure 4.2: Assignment of texture components to booleans required for the cellular automaton algorithm proposed by Dobashi et al. [4].

As shown in Figure 4.2, each of the color components represents one of the Boolean values described in Dobashi's paper [4]. The red channel is used to store humidity, the green channel is used to store the cloud bit and the blue channel is used to store the activation bit.

It is necessary to use eight bits per channel (even if we deal with boolean values), because most hardware does not support logical operations like AND (&), OR (|) or shifts, which makes it impossible to process the data in an efficient way. Hardware which is DirectX 10 certified and supports Shader Model 4.0 allows such operations, which means that a significant amount of graphics memory can be saved by using only one bit per *hum, act* and *cld* value. Nevertheless, you should keep in mind, that during rendering it is necessary to extract the *cld* value and this may slow down the vertex shader. Such an implementation would only work on DirectX 10 compatible hardware.

Since floating point values are used in the shader to access texture components (usually bounded by [0.0, 1.0]), 0.0 is defined as *FALSE* and 1.0 is defined as *TRUE*. To avoid filtering problems (even if filtering should be set to point/nearest neighbor) for texture fetches from this texture values less than 0.5 are interpreted as *FALSE* while values greater or equal 0.5 are interpreted as *TRUE*.

A similar approach was already presented by Lutz Latta in his article Building a Million Particle System [11], where textures were used to store the current simulation state for particles. Even updates to the states were computed on the GPU by using the pixel shader.

4.2. Simulation Step Computation on the GPU

Two textures are used for the simulation step, one for timestep t_i and one for timestep t_{i+1}.

These textures are labeled $Volume_A$ and $Volume_B$. On startup, $Volume_A$ needs to be initialized – depending on the type of simulation (the naive approach without much control or the method presented in Section 4.3.2) it would be sufficient to initialize all three components of the texture with zero. If the naive approach is used, some activation bits should be set at random (or user specified) positions, enabling cloud formation. It will also be necessary to set humidity values. A probability value of about 0.3 to 0.4 for the humidity bit to be set is recommended for good results.

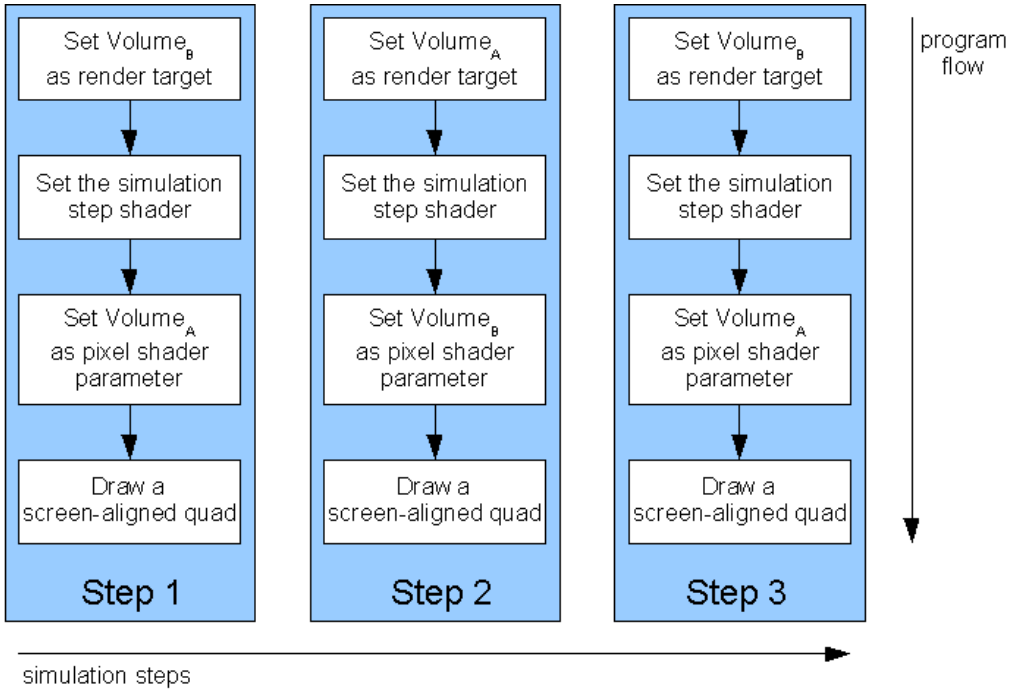


Figure 4.3: Program flow for simulation steps t_i, t_{i+1} and t_{i+2}.

Figure 4.3 shows how the simulation step is performed on the GPU. The two textures Volume_A and Volume_B are used in ping-pong fashion. In step one Volume_B is used as render target while Volume_A is used as input for the pixel shader which performs the simulation step computations. For the second simulation step, the textures are switched and Volume_B (which currently stores the results of the first simulation step) is used as input for the shader while Volume_A is set as render target. For the next simulation step, textures are switched again since Volume_A now stores the results of simulation step two and VolumeB is no longer needed, so it can be set again as render target and the result of step four can be written to it.

4.2.1. Using the Pixel Shader for Simulation Step Computation

As shown in Figure 4.3, a screen-aligned quad is rendered to perform the simulation step. The screen-aligned quad renders a texture with the specified dimensions (i.e. 2048x1024, as shown in Figure 4.1) and thereby every single pixel of the volume is written. The resulting texture of the previous step is used as input for the pixel shader. Within the pixel shader the whole work is done. The rules proposed by Dobashi et al. [4] are implemented within there.

4.2.2. Cloud Growth Simulation

```
//get current states at actual map position
float4 vLastStep = tex2D(SamplerLastStep, In.TexCoord.xy);

//compute new humidity value with hum(t+1) := hum(t) and not act(t)
//note, that the humidity value is written to the r-component
if((vLastStep.r >= 0.5) && (vLastStep.b < 0.5)) {
  vResult.r = 1.0;
}
else {
  vResult.r = 0.0;
}
```

Code sample 4.1: Implementation of equation 2.1 - $hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i)$ and not $act(i,j,k,t_i)$

Code sample 4.1 shows how the rules defined as equations 2.1 to 2.3 can be implemented with the high level shader language (HLSL) [13] to be computed by the GPU. As already defined above, the red channel is used to represent humidity. *vlastStep* is read from texture and represents the result of the last simulation step for the current position (note, that this code fragment is performed for every single pixel on the texture). According to equation 2.1, the humidity bit should only be set when it is already set (from previous steps) and when the activation bit is not set. To avoid filtering problems (filtering could also be disabled – which means that point/nearest neighbor filtering is used) as well as floating point precision problems, tolerance values are used to interpret whether a 'bit' is set or not. *(vlastStep.r >= 0.5) && (vLastStep.b < 0.5)* evaluates only to true if a humidity bit is already set in the last step and no activation bit existed in this step. If the condition evaluates to true, 1.0 will be set for the r channel, which means that humidity exists; else the red channel will be set to false. This equals exactly Equation 2.1.

In a similar way it is possible to implement Equation 2.2. For Equation 2.3, it will be necessary to read neighbor values from the same slice as well as from other slices. Of course this can also be realized with a simple texture lookup. It is only necessary to compute the correct pixel position. For the reference implementation, a few boundary conditions were not implemented because they would slow down the shader and the states which may result because of the missing conditions are not noticeable.

4.2.3. Cloud Extinction and Humidity Regeneration

For the implementation of Equations 2.5 to 2.7, some random values are required. Since today's GPUs are not capable of generating random numbers, a workaround is necessary.

For the reference implementation, a precomputed noise texture (Figure 4.4) is used to read random numbers. This texture uses only a single channel with eight bits, which means that numbers within [0; 255] can be encoded. Within the pixel shader, these values are then interpreted as values within [0.0; 1.0].

To avoid that the same "random" numbers are read again and again, some CPU-generated random numbers are used as offset parameters for x and y and set as parameters for the simulation step shader. These offsets are added to the current texture coordinates when reading the noise map to get different random numbers. Of course the offset values need to be updated before a new simulation step will be performed.

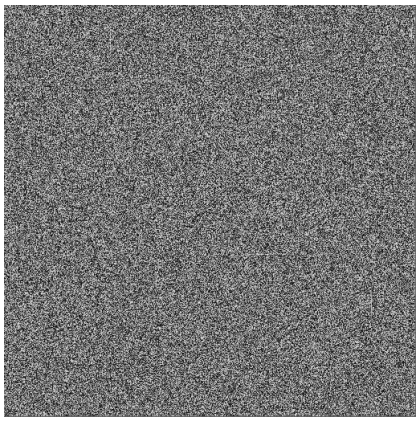


Figure 4.4: Precomputed noise map which is used to fake random number generation on the GPU.

Note, that the noise map would need to have at least double width and height as the cloud volume texture for best efficiency. Using these dimensions, it is possible to divide the input texture coordinates by two (which leads to a maximum value of 0.5) and adding x and y offsets within [0; 0.5] to make sure that a wide range of random numbers can be used. Since this would require a large amount of graphics card memory (an eight bit 4096x2048 texture would require 8 MB) it is also possible to use *WRAP* or *MIRROR* as texture addressing mode, which allows using a smaller texture with random values. See the DirectX SDK documentation [13] for more details on texture addressing modes.

Now that it is possible to access random numbers within the pixel shader, it is very easy to implement equations 2.5 to 2.7.

```
 //compute cloud extinction
 vNoise = float2(In.TexCoord.x / 2.0 + fNoiseOffsetExtX, In.TexCoord.y / 2.0 +
fNoiseOffsetExtY);
 fNoiseValue = tex2D(SamplerNoise, vNoise).a;
```

```
if(fNoiseValue <= fExtinctionPropability) {
  vResult.g = 0.0;
}
```

Code sample 4.2: Implementation of equation 2.5 - $cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i)$ and IS(rnd > pext(i,j,k,ti))

As shown in code Sample 4.2, where *fNoiseValue* is read from the noise texture using the randomly generated noise offsets, the cloud value will only be set to false if the noise value (which is our random number) is less or equal than a specified extinction probability value (which is assumed to be constant for this sample; in the final shader different values for the cloud field will be used instead).

According to Equations 2.1 to 2.3, which are executed before code sample 4.2, the cloud bit may be currently set or unset. If it is not set in the last step, Equations 2.1 to 2.3 would lead to the situation, that it is not set now. This would not be a problem even if the condition in sample 4.2 evaluates to true, because the cloud value would just be set again to false. So the only thing to care about is what to do if the cloud bit is currently set to true (1.0). In this case, if the condition evaluates to true, the cloud value will be set to false and extinction occurs.

In a similar way, equations 2.6 and 2.7 can be implemented. The whole HLSL code for simulation of the volume can be found in Appendix A, but keep in mind that so-called region volumes are used to control cloud formation and motion as well as extinction. To understand the code it is recommended to read Chapter 4.3 first.

4.3. Controlling Cloud Formation and Motion

Until now, the control of cloud formation (growth simulation) and motion (advection by wind) is very limited. Advection by wind was not mentioned in the previous section, since the proposed method will be much more flexible than equations 2.8 to 2.10 as proposed by Dobashi et al. [4]. In the following subsections, there we will first shortly discuss what problems may occur without motion control mechanisms, then a real-time implementation of the approach presented by Dobashi will be described to perform motion control.

4.3.1. Problems without Formation and Motion Control

The simple volume simulation step using only Dobashi's transition rules [4] (as mentioned in Section 2) has some serious drawbacks.

It is theoretically possible to control formation and extinction of clouds by adjusting their probability values (which may be constant for the whole field) for every simulation step. If this is not done, after a few steps the clouds will either completely disappear or cover the whole sky (assuming that humidity is uniform over the volume). To prevent this, an analysis of the volume would be necessary after every simulation step in order to decide how the new probability parameters should be chosen. Analyzing the volume is expensive and would completely overcome the advantage of the fast cellular automaton algorithm. Even if the volume can be analyzed in an efficient way, it would still be hard to stabilize the simulation to generate a nearly constant cloud cover.

4.3.2. Controlling Cloud Motion with Ellipsoids

Dobashi et al. proposed in their paper [4] to use ellipsoids to define regions where clouds may frequently appear and where extinction occurs. In fact, these ellipsoids are just used to define the above mentioned (see equations 2.5 to 2.7) probability values p_{act}, p_{hum} and p_{ext}. Vapor probability and phase transition probability (hum and act) are assumed to be higher at the centers of ellipsoids than at the edges. Extinction probability is assumed to be lower (nearly zero) at the center of the ellipsoids (since extinction does not usually start in the center of a cloud) while the probability is higher on the edges. This method negates the performance gain which could be achieved by using bitfield manipulation functions as described by Dobashi et al. It also highly increases computation time for ellipsoid and probability computations. To overcome these drawbacks, an extension to Dobashi's idea, called *region volume*, will be presented in the next section, which allows using ellipsoids for motion control with marginal extra effort.

4.3.3. Using Region-Volumes to Control Formation and Motion

The main idea behind region volumes is that ellipsoids should be encoded in a 2D texture the same way as it is the case for the volume map. This allows performing probability lookups in a simple and efficient manner. In the following subsections, the theory of region volumes and the integration into the simulation shader will be discussed.

4.3.3.1. Introduction to Region-Volumes

For the purpose of this thesis, a region volume is defined as a 2D texture that is organized in the same manner as the cloud volume maps (see figure 4.1). This map is used to store probability distributions for a series of

ellipsoids as volume slices. To do this, a texture with at least one channel with eight bit precision is used, and probability values are encoded in the range 0.0 (on the surface of the ellipsoids and also outside of them) and 1.0 (in the center of the ellipsoids). This probability distribution is constant for each ellipsoid within the ellipsoid map. The values are then downscaled using different scalars to get probability values for humidity regeneration and phase transition. For extinction, an inverted and downscaled version of the values stored in the region volume is used. This ensures, that extinction occurs mainly on the surface of clouds and not in their center, which would look unnatural.

4.3.3.2. Encoding Ellipsoids on 2D textures

Before it is possible to generate a region volume, it is necessary to compute ellipsoid probability values and encode them into 2D textures as slices (as can be seen in Figure 4.5 and Figure 4.6), since it would be much more efficient to do this once at application startup than for every simulation step within the shader. In a preprocessing step, a few textures of different size that encode ellipsoid probability distribution are computed. The number of necessary textures depends mainly on the variety of clouds, but since memory requirements are relatively low for those textures (for example 64KB for a big ellipsoid with dimensions of 64x32x32), it is possible to create a high number of textures.

Probability values are, as mentioned above, always within [0.0, 1.0]. On the surface and outside of the ellipsoid, the probability value is 0.0, while it is 1.0 in the center of the ellipsoid. The surface points of the ellipsoids can be computed with the well known ellipsoid formula, as can be seen in equation 4.1. For pixels within the ellipsoid, a linear interpolation between the center and the surface (in direction from center to the current point) is performed.

An example of a texture, which encodes the probability distribution for a single ellipsoid can be seen in Figure 4.5.

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}-1=0$$

Equation 4.1: Ellipsoid equation

For simplicity, the number of slices for these textures equals always the number of slices which are used for the cloud volume, even if the ellipsoid's height is smaller. Of course, it is also possible to encode all slices side by side, as long as the maximum texture width that is supported by the target hardware is not exceeded. This would slightly simplify addressing and

therefore improve performance.

Since not every GPU supports non-power-of-two texture dimensions (which are dimensions not within 2^x for values of x larger or equal to zero), and other GPUs that support non-power-of-two texture dimensions may suffer under performance problems if such textures are used, it is recommended to use only textures with power-of-two dimensions, even if space is wasted. For example, an ellipsoid that has a width of 26, a depth of 18 and a height of 20 is encoded on 32 slices (which equals for this example the height of the cloud volume) where every slice has dimensions of 32x32 (since 26x18 is extended to the next existing power-of-two dimension).

Figure 4.6 shows a 3D sketch that explains how ellipsoids are encoded into volume slices. Every slice stores probability values which are then used to render a region volume.

Figure 4.5: 2D texture used to encode probability distribution for a single ellipsoid using 32 slices. Black represents a probability value of 1.0, white represents a probability of 0.0.

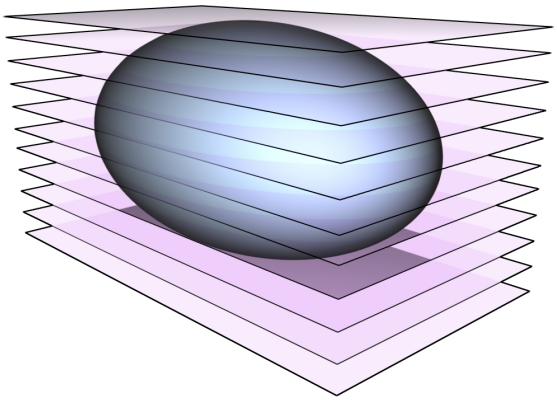


Figure 4.6: 3D sketch that shows how ellipsoids are encoded in volume slices.

4.3.3.3. Building a Region Volume

To create a region volume, it is necessary to define position, size and orientation of each ellipsoid to be placed inside the volume. These parameters can be stored in a list structure that is updated before the next simulation step is computed. For example, the positions of the defined ellipsoids are slightly modified to simulate advection by wind. In the next subsection, a few extra attributes will be introduced, but the mentioned main attributes position, orientation and size (or ellipsoid texture index) are indispensable for region volume generation. For initialization, it is possible to simply use random position, orientation and size attributes, but it is recommended to avoid placing all ellipsoids at the same time, since this would lead to unnatural results.

After the list structure that stores ellipsoid information is initialized/updated, it is used to render the ellipsoids into the region volume. This is done by rendering one slice of the volume after another. For each slice, the complete list structure storing ellipsoid information will be iterated and corresponding ellipsoid slices will be rendered to the current region volume slice using alpha blending with a "maximum" blending function.

After all iterations have completed, the region volume stores all necessary ellipsoid probability distribution values and can then be used to control the

cloud volume simulation step.

Figure 4.7 shows a sketch of a region volume that stores probability distribution values for a number of ellipsoids.

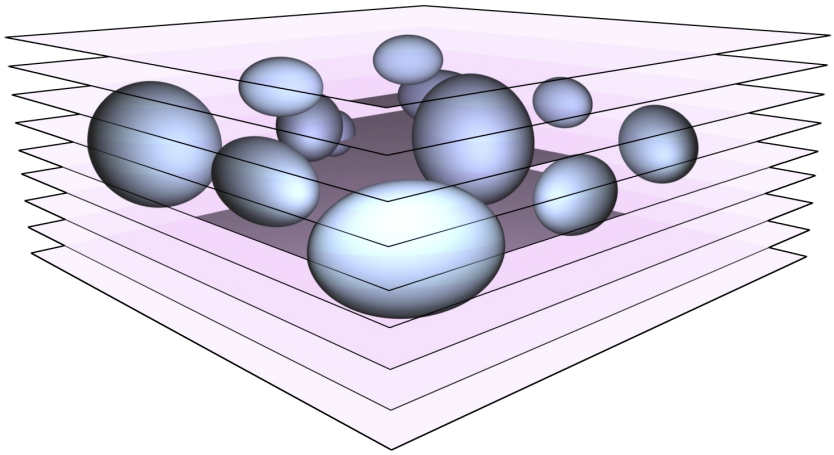


Figure 4.7: Region volume sketch where a few ellipsoids are encoded into the slices.

Depending on the number of ellipsoids that will be rendered into the region volume, this step can be expensive and it is useful to use hardware instancing for drawing instead of a separate draw call for each ellipsoid slice. Of course, it is also possible to split the necessary work for region volume generation over multiple frames by rendering only one region volume slice per frame. Since region volume updates are usually only necessary every few seconds, this optimization is very useful to avoid inhomogeneous rendering times for frames because there are for example 32 frames (for 32 region volume slices) used to perform the computations, where rendering time for each frame becomes marginally higher instead of a single frame where the rendering time differs significant from the other frames.

4.3.3.4. Definition of Necessary Parameters

First, to get a stable simulation with a nearly constant user-defined cloud cover, it is necessary to define some parameters which will be used for region volume generation and which can be adjusted by the user in an intuitive and easy way.

Cloud density specifies how many ellipsoids should be used for region volume rendering; note that some of them will not be within the visible area; the reason for this will be explained during description of the *region boundings* parameter.

Density adaption per step specifies how many ellipsoids can be removed from or added to the simulation in a single step; this allows controlling how fast the cloud cover should change without increasing the simulation speed (which would affect i.e. also the influence by wind).

Average cloud size is used to specify the default size of region sprites; this allows generating different types of clouds.

Cloud size variance is used to define the maximum aberration from the average cloud size parameter.

Average lifetime specifies how long an ellipsoid should exist before it is removed from the simulation. To avoid unnatural cloud extinction, it is recommended to reduce the size of an ellipsoid shortly before the lifetime ends. This ensures that extinction occurs correctly and the corresponding cloud mostly disappears before the ellipsoid is removed from the simulation.

Lifetime variance allows to set a variance for the average lifetime parameter.

Region boundings are used to define the area where ellipsoids can be placed. Whenever an ellipsoid is out of this area, it will be removed from the list. The area should be generally larger than the cloud volume. This makes sure that clouds can slowly drift into the visible area and also allows the user to perform fast movement without influencing simulation stability. This method furthermore has the advantage that the user could return to clouds which drifted already out of the volume. This may be especially interesting for flight simulations where relatively fast movement and frequent direction changes occur.

Figure 4.8 shows how the region boundings should be used to make sure that enough ellipsoids are outside the volume which can then move into the volume when it is necessary, i.e. if fast camera movements or high wind speeds are used. Ellipsoids which move outside the region boundings are removed from the simulation and (if necessary) a new ellipsoid is created on

a random position inside the region boundings instead.

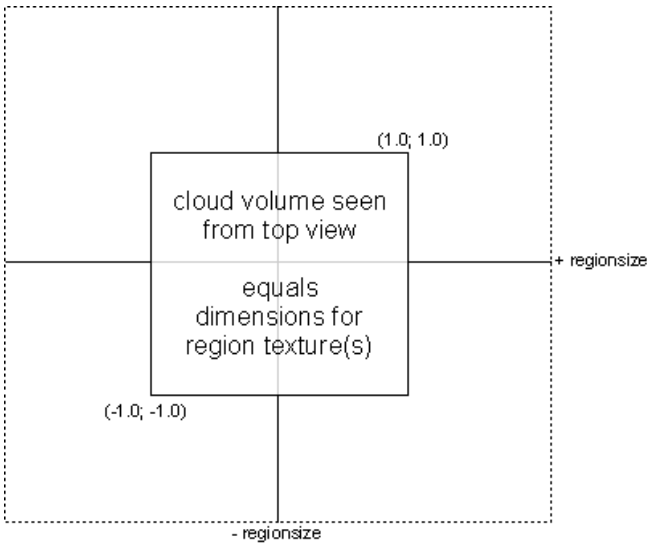


Figure 4.8: Region boundings defined by a regionsize parameter.

Wind vector is used to specify the direction of ellipsoid movement as well as the speed of movement (by using the vector's length). Note that a small variance should be included to make sure that not every ellipsoid moves with the same speed in the same direction because this would not look very realistic. Optionally, it would be possible to use multiple wind vectors to create a more realistic flow simulation of the volumetric data, as can be seen in Figure 4.9. Such vectors could easily be generated in an algorithmic way or it would even be possible to use real-world data which can be read from various sites on the internet. Nevertheless, this is outside the scope of this thesis and should just be mentioned to highlight the potential of the method.

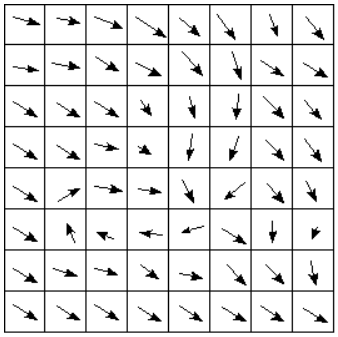


Figure 4.9: Two-dimensional array consisting of 8x8 elements which is used to store wind direction vectors. Note that the vectors differ also in length, which is used to model different speed of wind for local areas.

4.3.3.5. A Simple Method to Achieve a Stable Simulation

To get a stable simulation it is necessary to keep the amount and average size of ellipsoids nearly constant. To achieve this, a few simple rules are used for region volume rendering:

1. If there are fewer ellipsoids active than specified using the cloud density parameter, add new randomly generated ellipsoids to the ellipsoid list (which is just a structure to store an age/lifetime component, two components for position and two components for scaling in x and y direction. It may also be useful to store an orientation attribute). Never add more ellipsoids in a single step than defined with the density adaption per step parameter.

2. If there are more ellipsoids active than specified with the cloud density parameter, remove randomly selected ellipsoids from the list or choose the ellipsoids to delete depending on their current position or their age.

3. For every simulation step, decrease the lifetime of all ellipsoids by one. Whenever the lifetime is zero, remove the ellipsoid from the list.

4. For every simulation step, add the wind vector to each ellipsoid's position; also add a small random offset to improve animation quality, since it does not look very impressive if all ellipsoids move exactly with the same speed along the same direction (which is especially a problem for environments where only a single wind vector is used instead of an array of vectors).

5. If any ellipsoid is out of the the defined bounding rectangle, remove it from the list since it can be assumed that it is so far away, that it's lifetime would expire before it can get back into the volume area. This idea could also be used to automatically remove ellipsoids which are inside the region boundings, but have only less lifetime left and it would not be possible for them to get back into the volume area, even under optimal conditions.

Figure 4.10 to Figure 4.12, which are shown at the end of this chapter to demonstrate how the described parameters can be used to control the appearance of clouds, are generated with the reference implementation of the proposed method. They show 32 volume slices which are additively blended, viewed from the top, which leads to a rough overview of the could field. In Chapter 7, a method for performance improvement based on such additivly-blended maps will be introduced. The map can also be used to render shafts of light, as will be shown in chapter 7 too.

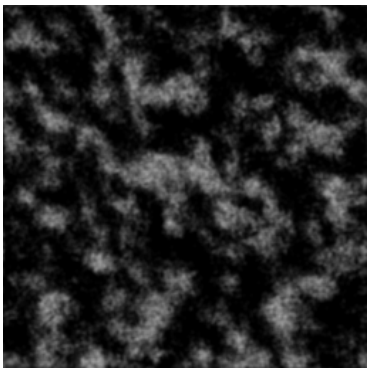


Figure 4.10: 32 additivly blended volume slices viewed from the top. A relatively high number (about 500) of small ellipsoids were used for rendering.

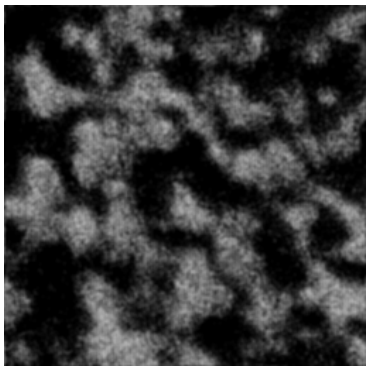


Figure 4.11: 32 additivly blended volume slices viewed from the top. Medium sized ellipsoids were used for rendering; their number was slightly decreased to avoid too high cloud cover.

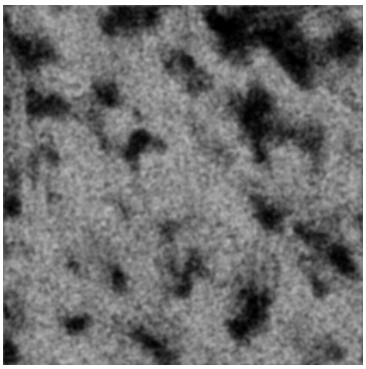


Figure 4.12: 32 additive blended volume slices viewed from the top. Very large ellipsoids were used for rendering; the number was significant reduced to avoid too high cloud cover.

4.4. Cloud Field Smoothing

As already described by Dobashi et al. [4], it is necessary to smooth the generated volume to produce a continuous density distribution, which is much more realistic than binary values. Equation 2.11 is slightly modified to perform cloud volume smoothing in real time using the GPU. To do this, it is necessary to copy the volume to another texture and since most graphics cards only support vertex texture fetches from floating point textures (which are necessary for rendering the clouds), such a texture is used as rendertarget. The dimensions are the same as for the volume textures.

Smoothing is performed using a simple pixel shader that computes an average value using for each voxel all neighbor voxels. Since this requires accessing successor and predecessor slices, the most simple solution is to copy and smooth one slice of the texture after another. Predecessor and successor slice bounding vectors are supplied to the shader as constant parameters to keep addressing as simple as possible.

Unlike described by Dobashi [4], smoothing is only computed over space, not over time, which means that no previous simulation step results are involved. This is necessary to keep memory consumption low and performance as high as possible. Since an interpolation between two volume textures is performed during rendering, smoothing over time is implicitly done.

5. Cloud Shading

5.1. Types of light scattering

Clouds are usually lit by the sun as well as by scattered light from the atmosphere. Shading computations should ideally be performed for every single cloud particle in the volume. For this task, it is necessary to take attenuation due to light traveling through the volume into account. According to Nelson Max's paper [15], single scattering only leads to a useful approximation if the albedo or the density is low. Since this is usually not valid for clouds, they may be unnaturally dark on some positions when *single scattering* (which is also called forward scattering) is used.

There are two ways to solve this problem. First, it is possible to simply introduce an ambient value, which prevents clouds from getting too dark. Even if it is not physically correct, the results are usable. This is also the fastest method to avoid the problem. Second, since true *multiple scattering* is computationally expensive and even not necessary for most scientific visualization applications, it is usually sufficient to concentrate on directions near the light vector. Harris [14] proposed such a scattering computation method, called *multiple forward scattering*, to achieve this. This method is also used in [17]. The method proposed in this paper allows using both methods with a higher accuracy than is achieved with the technique introduced by Harris [14]. For the reference implementation, single scattering with an ambient value was used for performance reasons and the results are admissible.

5.2. Cloud Shading on the GPU using an Inverse Shear-Warp Approach

The technique for computation of attenuation for every cloud particle is based on the method proposed by Dobashi et al. [4]. Instead of billboards, the volume slices themselves are used and the expensive frame buffer read back, which requires to copy the frame buffer to system memory for shading of each single particle, is not necessary for the proposed algorithm.

To perform cloud shading computations, a set of separate shading textures (which

are also 2D textures where slices of the volume are encoded) is used. The textures can differ in dimensions in relation to the cloud volume textures. If they share the same dimensions, shading information for every single cloud particle can be encoded. For performance improvements, it would also be possible to reduce the dimensions to half or a quarter of the cloud volume. Harris [17] also proposed to use light volume dimensions with half the resolution of the cloud volume. Compared to the method proposed by Harris, where accuracy depends on the light direction vector and which may waste a relatively high number of shading map pixels (see also Section 2), the proposed shear-warp based approach is much more efficient and uses all pixels of the map to store shading information for corresponding cloud particles.

The method presented in this thesis works in a similar way like the well-known shear-warp volume rendering [21]. To my knowledge, it has never been used for shading computations in real-time environments before.

There are at least three special cases (depending on the light direction vector) that have to be considered. For light vectors in a range of [0, 45] degrees (where zero corresponds to a direction vector of <0.0, -1.0, 0.0>, the light comes mostly from above the volume and therefore, shading is processed based on volume slices in y-direction (where the y-axis represents height in a left-handed coordinate system).

Case one is shown in Figure 5.1. This slice orientation and iteration is used for light direction vectors that differ less than or equal to 45 degrees from the vector marked red (which is defined as <0.0, -1.0, 0.0>).

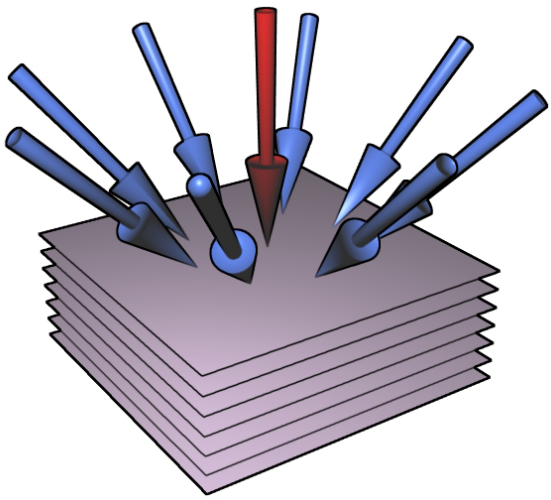


Figure 5.1: Case one for slice iteration during cloud shading. This case is used if the light direction vector differs not more than 45 degrees from the red marked direction vector.

As can be seen in Figure 5.1, shading is computed in a top-to-bottom order by starting at the slice which is farthest away from the ground iterating down to the slice that is nearest to the ground.

Shading is computed in a ping-pong rendering approach, where two textures are alternately used as render target and input texture to the pixel shader. These two 2D textures are organized in the same way as the cloud volume texture, which means they are organized in slices. The textures need only a single channel and are used as render target. Most hardware does not support rendertargets with an 8bit alpha component only (A8), so it is either necessary to use an (A8)R8G8B8 texture format or (which is recommended for optimal quality) a 32 bit floating point format instead. Nevertheless, even though hardware like NVIDIA's GeForce 6000 series supports 32 bit floating point textures as render targets, bilinear filtering for these textures is not available. If older hardware needs to be supported, an (A8)R8G8B8 target, where only one component is used, would be preferred. The GeForce 8000 series allows bilinear filtering for FP32 (floating point texture format with 32 bit precision for a single component) and therefore it is today possible to perform shading computations with high precision entirely on the GPU.

Computation of the attenuation ratio for each voxel in the volume is performed as follows:

- The slice (area on one of the two ping-pong rendering textures) which is farthest away from ground is initialized with 1.0 (on ping-pong texture A), since there can be no occluder in front of the first slice and therefore particles in this height are fully lit. Because slice 0 was assumed to be the slice with the smallest distance to the ground in Chapter 4, this assumption is kept for the shading computation, which means that the slice initialized as 1.0 has the highest index.
- Switch ping-pong textures. Set the texture that was used as render target before now as input texture for the shader; set the other texture as render target.
- Based on the light direction vector, offsets for shifting the slices during iteration, are computed. For a height difference of one between the slices (which should usually be the case, since the voxels are aligned in a regular grid structure), the y-component of the light vector needs to be one to get correct offsets. To achieve this, all three components x, y and z are divided by the unsigned y-component. Now, x and z components store the offsets for shifting the slice in voxel space. Since the volume is stored on a texture and texture coordinates for the quad which is used to render slices are within [0,1] for both dimensions, the offsets need to be divided by the texture dimensions to transform them to texture space.

- For each slice k from n-2 (the top slice is already initialized) down to 0:
  - set viewport parameters to make sure that only the part of the rendertarget where slice k is located is written.
  - compute bounding vectors $<x_{min}, y_{min}>$ and $<x_{max}, y_{max}>$ in texture space for slice k+1 (which is the predecessor slice). Supply these bounding vectors to the shader as constant parameters and also supply the computed offset vector for slice shifting. One of the ping-pong-textures (the one that stores the already computed slice k+1) is also supplied to the shader, as well as the volume texture itself.
  - Render slice k into the texture on the correct position.
    - the pixel shader copies the attenuation values from slice k+1 (which are stored on the supplied ping-pong texture) shifted by the computed offsets to the current slice. New attenuation that occurs from volume slice k+1 is also considered and shifted in exactly the same way than the previous shading slice. This leads to a propagation of attenuation ratio through the volume, as can be seen in Figure 5.2.
  - switch ping-pong textures again.
- Each of the two ping-pong textures now store n/2 volume slices. To get the final shading map, the two textures need to be combined. This can either be done using alpha blending (if supported on the hardware for floating point textures) or using a simple pixel shader, which nevertheless has the drawback that a third texture is necessary (which is used as rendertarget) while the ping-pong textures are supplied to the shader as constant parameters.

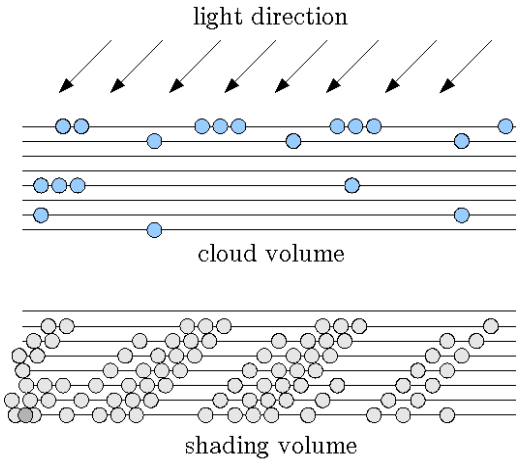


Figure 5.2: Profile of volume slices and corresponding shading slices. The shading information is propagated down in direction of the light vector. Every cloud particle hit absorbs some amount of light and therefore darkens the shading map on the light direction path on the successor slices.

The two other special cases are necessary to avoid that the offset to shift slices

becomes too large. To do this, slices are simply oriented on a different axis, as it is also necessary for the shear-warp volume rendering algorithm. Orientation for different light vectors are shown in Figure 5.3 and Figure 5.4. As can be seen in these figures, it is possible to use a slice orientation always for two sides. It is just necessary to flip the iteration order from slice 1 (while slice 0 is initialized with 1.0) to slice n-1.

The three special cases shown in Figures 5.1, 5.3 and 5.4 together cover light direction vectors for the whole hemisphere. Popping artifacts which can occur when switching from one slice orientation to another are marginal and can't noticed by the user when interpolation is used.

To simplify texel addressing for cases two and three, it is recommended to generate cloud volume maps that organize slices in the same way as the shading map slices are organized. This would lead to extra overhead and increase also memory consumption for the additional (and redundant) volume maps. On the other hand, it can increase performance during cloud shading, which may be done more often than volume simulation steps. Furthermore, the quality will be better since bilinear filtering is only possible (or useful) if the slice organization on the shading map corresponds to the slice orientation on the volume map. The two copied volume maps can be created in a single rendering pass using two render targets simultaneously. The reference implementation usees this method.

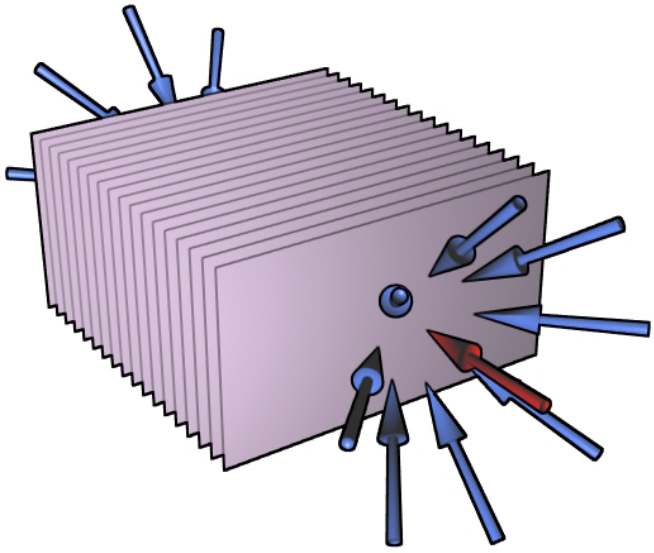


Figure 5.3: Case two for slice iteration during cloud shading. This case is used if the light direction vector differs not more than 45 degrees from the marked direction vectors. Depending on which one of the two red vectors are used, the iteration order for slice rendering has to be switched.

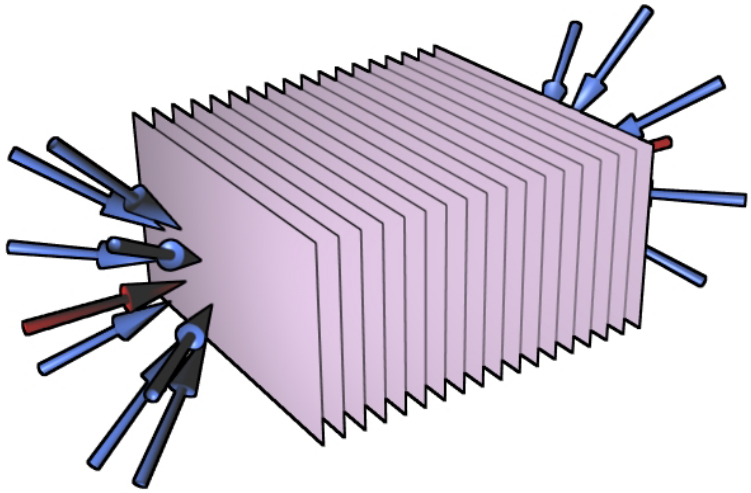


Figure 5.4: Case three for slice iteration during cloud shading. This case is used if the light direction vector differs not more than 45 degrees from the marked direction vectors. Depending on which one of the two red vectors are used, the iteration order for slice rendering has to be switched.

Depending on the light direction vector (which is used to decide which slice orientation is useful), it may be necessary to perform many draw calls (i.e. 256 or even 512). Even if the shading algorithm itself is very fast, too many calls would slow down the simulation because the application may become CPU limited, as explained in Section 2.2. To avoid this, and since shading computations are not necessary for every frame, it is recommended to split the work over several frames. I.e. rendering of one shading slice per frame will practically not influence the framerate.

6. Cloud Rendering using Point-Sprites

Modern GPUs support a primitive type called "point sprite", which has some serious advantages compared to classical billboard approaches which are mostly rendered using two screen-aligned textured triangles (forming a quad or at least a rectangle).

6.1. Advantages of Point Sprites Compared to Classical Billboards

Usually, as already mentioned, billboards consists of at least two triangles, which would lead to 6 vertices that have to be transformed for rendering. For example one million particles, would lead to 6 million vertices which have to be transformed by the vertex shader in a way that makes sure that the billboard is always oriented towards the viewer. Since we need to group billboards for rendering (remember Section 3; the number of draw calls needs to be as small as possible) this leads to a relatively expensive shader which can end up as a bottleneck.

A GeForce 6800 Ultra is able to – according to NVIDIA – process up to 600 million vertices per second, which would theoretically lead to 100 million billboards when non-indexed data is used, so one million particles should be processed about a hundred times per second. In practice, on a GeForce 6800 Ultra, we have found that rendering only about 100.000 particles that way would make the application vertex transform limited and the framerate would fall to 10-20 frames per second.

To reduce the number of vertices which need to be transformed, indexed polygons can be rendered. Instead of six vertices per triangle, four vertices would be sufficient, which reduces the number of vertex transforms by about 33%.

Nevertheless, the necessary computations could also be achieved with a single transformation per cloud particle, which is the optimal scenario. To achieve this, point sprites were introduced, which are (unlike other geometry) always oriented towards the viewer. This makes it possible to transform only a single vertex (the center position) while the rest is performed in screen space coordinates, which means that the size in pixels need to be computed in the vertex shader based on

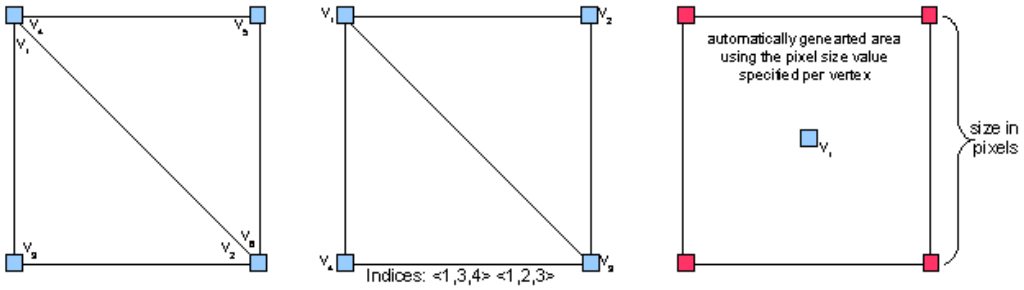


Figure 6.1: The left quad is generated using 6 vertices, where v_1/v_4 and v_2/v_6 share the same coordinates. The center quad is generated with only four vertices using an index buffer with groups of three indices that specify which vertices should be used to form triangles. The right quad is drawn with a single vertex (which is optimal) using point sprites. The size of the sprite is specified as component of the vertex data stream.

the distance to the viewer.

Figure 6.1 shows an overview of the above mentioned methods.

Of course it is also possible to put textures on point sprites and to use them in combination with pixel shaders, both of which is necessary for rendering cloud particles with the proposed method. Because they are already supported on most (consumer) graphics hardware for the last years and they do not have any disadvantages (with respect to this thesis), they are used to render cloud particles to the screen.

Nevertheless, note that there may be at least a theoretical problem since the size of point sprites is often limited. The GeForce 6800 i.e. supports point sizes of up to 8192, which is already more than sufficient for cloud particle rendering (since cloud particles are usually relatively small as they are seen from ground and even if we were directly in front of them, screen resolutions of that size are not practical on most systems today).

6.2. How Clouds are Rendered

Cloud particles are rendered using a splatting [22] based approach. As already mentioned in Chapter 3, it is necessary to keep the number of draw calls as small as possible. To achieve this, cloud particles will be grouped into a grid-structure of i.e. 32x32 particles which represent a part of the cloud volume.

Figure 6.2 shows how cloud particles are grouped together to dramatically reduce

the number of draw calls. I.e. the worst-case scenario for a volume with dimensions of 256x256x32 will be over 2 million calls. 16x16 groups will reduce the number of calls by a factor of 256; 32x32 groups will reduce the number of calls by a factor of

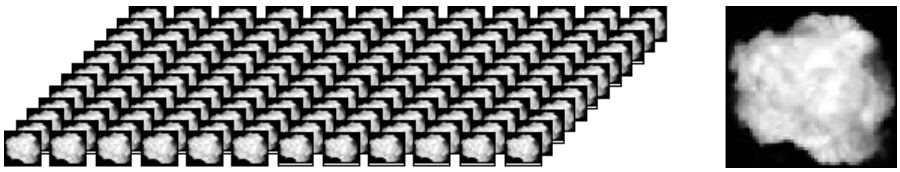


Figure 6.2: Grid of 16x16 textured point sprites which are aligned on a common plane. The right image shows the 64x64 pixel texture that is used to render the cloud particles. Note, that it is necessary to choose distance and size of the particles to let them overlap (which is not shown in this image), if this is not the case, the cloud field will have empty spaces which look very unnatural.

1024 – which leads to a maximum of 2048 calls for the above-mentioned volume. This can be further dramatically reduced by techniques described in Chapter 7.

Nevertheless, grouping introduces a new problem – when particles are grouped together they can't be sorted in a correct way (i.e. back to front) since it is not possible (because of parallelism) to instruct the GPU in which order the polygons in the vertex stream should be rendered. This can only be avoided by using a separate draw call for each particle, otherwise errors will occur.

The trick is to keep the errors small enough to avoid that they are noticeable for the viewer. The way to do this is by adjusting the grid size until the error becomes acceptable (according to the visual impression).

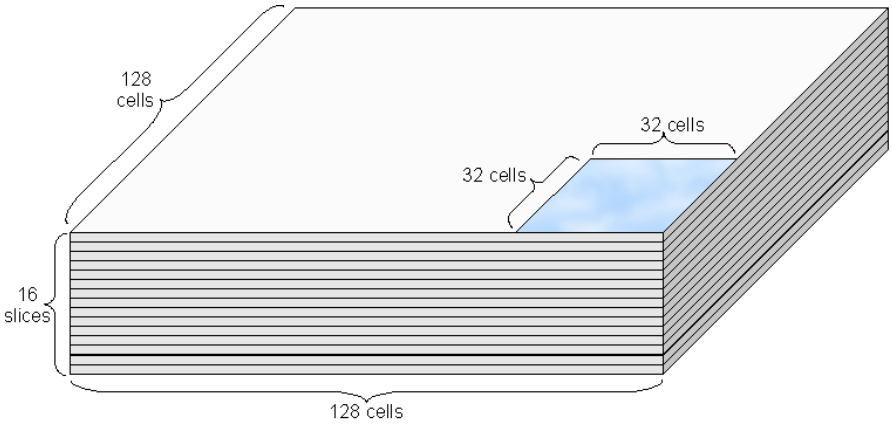


Figure 6.3: Segment of 32x32 point sprites (which represents 32x32 cells of the volume.

As can be seen in Figure 6.3, there is always a fixed number of point sprites used to represent a fixed segment of the volume. This has the advantage that only one vertex buffer is necessary and this one is re-rendered on different positions as often as it is necessary. Since the cloud particle value is not set for every single cell, it is necessary to decide within the vertex shader if the current point sprite should be drawn or not. To do this, a feature called *vertex texture lookup* is used. This feature, which was introduced with Shader Model 3.0 (at least on NVIDIA hardware), allows performing texture lookups within the vertex shader. Note that on some hardware restrictions may exist. I.e. on the GeForce 6800 Ultra it is only possible to use floating point textures, values from RGBA textures can't be fetched. To implement the algorithm on hardware which has only support for floating point textures it would be necessary to first write the cloud values to a floating point render target in a separate pass, which is in fact not a big problem, so it should just be mentioned here shortly.

The idea behind the method is very simple: every point sprite is read from the vertex stream and processed within the vertex shader. Shader parameters are used to specify which slide of the volume is currently processed and which offsets should be used to identify the correct segment of point sprites. The point sprite vertex stream itself stores position components for every sprite, which reaches i.e. for 32x32 sprites from `<0.0, 0.0, 0.0>` to `<31.0, 0.0, 31.0>`. Using this information it is possible to fetch the correct value from the volume texture to check if cld is *TRUE*. If this is the case, the point sprite's position component is transformed to post-perspective space by multiplying it with a WORLD * VIEW * PROJECTION matrix. Based on the position from the user, the size of the point sprite (which is also a vertex shader output component) is computed as well. If the result of the texture lookup indicates, that cld is *FALSE*, the point sprite can be thrown away and does not need to be further processed. As already mentioned in Chapter 3, the vertex shader takes exactly one vertex as input and writes exactly one vertex as output, so it is not possible to "discard" vertices or to add new geometry (except on hardware that supports so called *geometry shaders,* which can currently only be used in the new GeForce 8000 series and are relatively slow). To remove point sprites which should not be rendered, a simple trick is used. In fact, it is sufficient to output a position vector that is out of the view volume, for most purposes in Direct3D a negative z-component can be used. (Note that this is not always true, it depends on your projection transformation as well as on the viewport's MinZ and MaxZ components that are set when initializing Direct3D). Due to optimizations, the graphics card will automatically perform clipping and therefore stop a processing point sprite which is definitely outside the view volume. Figure 6.4 shows the idea of rendering clouds with point sprites.

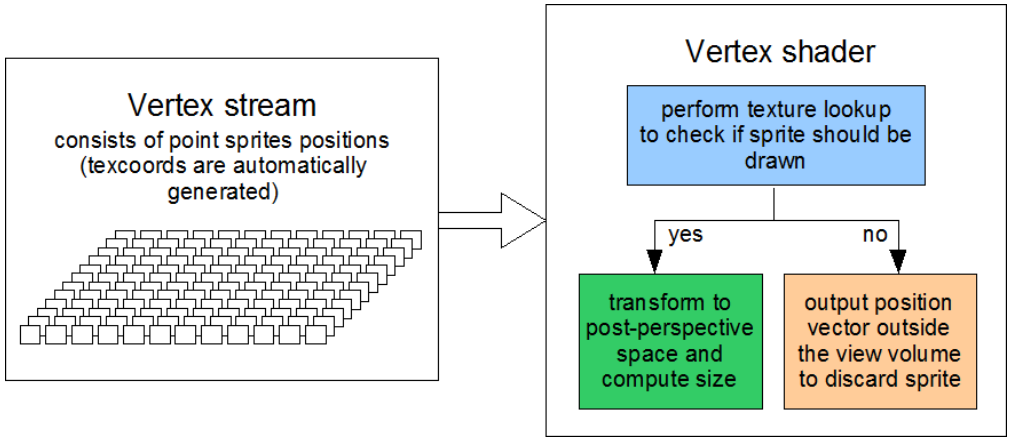


Figure 6.4: Each vertex in the stream represents a single point sprite. For each point sprite, the shader code is executed. A texture fetch is performed on the correct position within the volume to check if the sprite should be drawn. If no sprite is required on the position, it is 'discarded' by writing an output position outside the view volume.

6.3. Smooth Animation by Performing Volume Interpolation

According to the nature of the simulation step as described by Dobashi et al. [4], we can compute results only for discrete timesteps t_i, t_{i+1}, t_{i+2}, etc. Computation for i.e. timestep $t_{i+1.7}$ is not possible. This introduces a new problem, because modifications between two simulation steps can be highly noticeable.

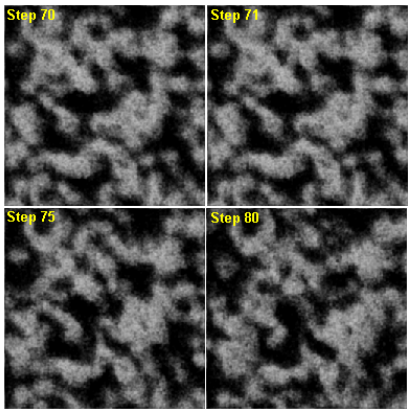


Figure 6.5: The four images shows the simulation step results starting at step 70 (such a high step is chosen to stabilize it, so that the cloud formation starting from the empty volume has nearly no influence). As can be seen, even in the next step there are visible changes which would lead to some kind of popping artifacts without interpolation. Step 80 already looks completely different to step 70.

This makes it necessary to not only use the latest simulation step texture (step t_i) for rendering the clouds – at least the texture of step t_{i-1} is also necessary to perform an interpolation for every single *cld* value. Due to the fact that it is required to store the previous timestep to perform the volume simulation (because it is needed as input texture for the shader), both textures are available anyway and can be used to perform the rendering step. In the reference implementation, a linear interpolation is used and it looks visually acceptable.

7. Improvements

7.1. Splitting Cloud Volume Computation and Shading over Multiple Frames

As already mentioned in Section 4 and Section 5, the proposed algorithm can easily be adapted for splitting the necessary computations for the volume simulation step as well as the shading step for computation over multiple frames.

Figure 7.1 shows two different configurations. The left one is quite simple and straightforward. Whenever it is necessary, a new simulation step and a new shading map are computed within a single frame. When the computation finished, volume A and B are switched and interpolation is performed again over several frames before a new simulation step (and shading step) is required. However, in the frame where the volume map and shading is recomputed, a noticeable drop in performance occurs.

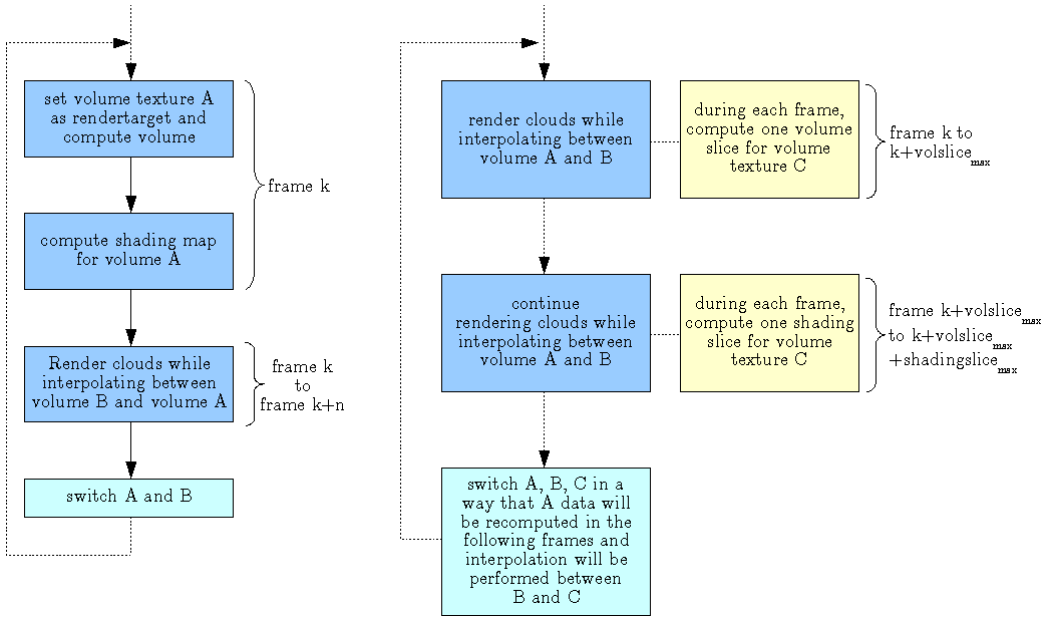


Figure 7.1: The left diagram shows a configuration where the complete volume simulation step as well as the shading step are computed in a single frame. The right diagram shows a configuration where a third texture set is used to split computations over multiple frames.

The right diagram shows a more sophisticated configuration for which a third set of textures (a cloud volume texture and a shading texture) are required. While the interpolation between volume A and volume B during cloud rendering is performed just like for the simple configuration on the left side of Figure 7.1, for each rendered frame, a single slice of the cloud volume texture C is computed. If all slices are updated, a single shading slice will be computed for each frame until the shading map is complete. Now, texture set A becomes the inactive one, which means that it is used for computation of the next timestep including the new shading map, while the interpolation for cloud rendering is performed between texture set B and C.

Splitting work over multiple frames has serious advantages compared to the naive approach of doing all the work in a single frame. First, on older graphics hardware, the simulation and shading steps could be expensive as framerate would drop to 15-30 frames or less, which will be noticeable for the viewer. Second, splitting the work allows better precision and even larger volumes can be used because the extra costs per frame are marginal and not noticeable.

If updates are required only every few seconds or even minutes, the work could be further split into subspaces of the slices, which allows processing i.e. only a quarter of a slices each frame.

7.2. Improvements for efficient cloud rendering

Since the presented cloud rendering method implies that a huge amount of point sprites need to be drawn (i.e. for a volume with dimensions of 256x32x256 it may be necessary to draw more than 2 million sprites), the biggest potential for improvement lies in reducing the number of necessary sprites without increasing the number of draw calls, while not increasing the overhead that may be necessary for the implementation of improvement approaches. In the following Sections 7.2.1 and 7.2.2, two methods to reduce the number of segments which need to be drawn are discussed.

7.2.1. Remove groups outside the viewing frustum

A trivial method to reduce the number of sprite groups is the well-known viewing frustum detection test, that can be used to test i.e. if a box, specified by two vectors, is inside, partly inside or outside the viewing volume.

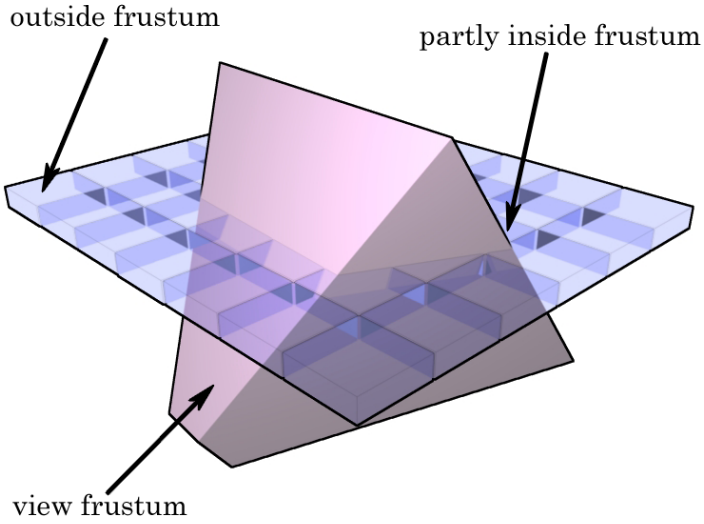


Figure 7.2: Viewing frustum detection test. Segments are classified to be inside, partially inside or outside of the frustum.

Figure 7.2 shows how this test can be applied to remove segment groups which are outside the frustum.

Segments are classified to be either outside the frustum, which means that they can be ignored since they are not visible, or (partially) inside the frustum.

The test could theoretically be performed for each point sprite group, but this would result in a relatively high CPU overhead and it would not be very efficient, because if a point sprite group is within the frustum, the probability that the group above/below is also within the frustum is high. This observation allows performing frustum tests for multiple groups at once by enclosing them in a common bounding box. For the reference implementation, all 32 slices are grouped together In such a common bounding box.

7.2.2. A fast method for empty space skipping

Given that for most situations the clouds will not cover the whole sky, there is great potential for performance improvements in reducing the number of necessary draw calls by efficient detection of areas where *cld* is set to 0.0 in the cloud volume.

In this subsection, a simple and efficient method to identify areas where no clouds exist is proposed.

As a first step, it is necessary to generate a density map from the cloud volume. To build such a map, all slices of the volume will be added together to create a top-view 2D image of the volume, as can be seen in Figure 7.3.

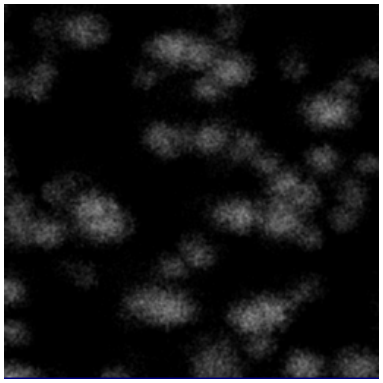


Figure 7.3: Density-map generated by adding all slices of the volume together.

For the rest of this subsection, it is assumed that the size of the point sprite group is 32x32 (of course, any other size can be used instead). The density map shown in Figure 7.3 with dimensions of 256x256 is split into 64 segments, which results in a segment size of 32x32, corresponding to the size of the point sprite group (as can be seen in Figure 7.4).

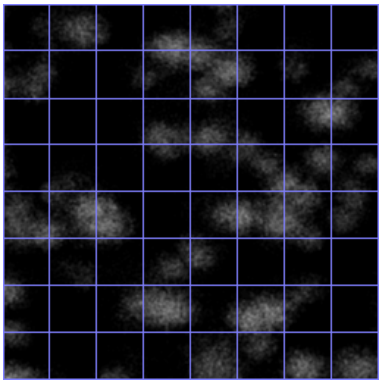


Figure 7.4: Density-map split into 64 segments.

Because the segment size corresponds to the point sprite group dimensions, for every segment on this map that is empty (which means that not a single pixel within this segment is greater than zero), the point sprite groups for all volume slices in this area don't need to be rendered (Figure 7.5).

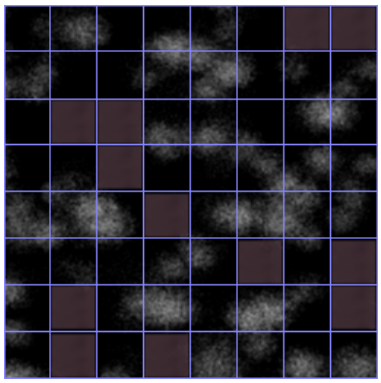


Figure 7.5: Density-map split into 64 segments where segments that are empty are marked.

Proving if an area is empty can be easily done by downscaling the density map using a maximum function until its size corresponds to the grid structure. That means the dimensions of the map are divided by two, which leads to the fact that 4 pixel on the original map are represented by a single pixel on the downscaled map. Because a maximum function is used, the highest value from the four pixels in the original map is selected. This downscaling operation is repeated until the dimensions correspond to the dimensions of the grid structure, which are 8x8 pixels for Figure 7.4.

The downsampled version stores the maximum density for each segment in a single pixel and segments where the corresponding pixel is zero can be discarded. Since the downsampled version is usually very small (i.e. 8x8 or 16x16 pixels), it can be copied to the system memory without performance loss, where it is used to draw only point sprite groups that are necessary.

Smaller point sprite groups result in a larger downsampled density map, which will allow a more precise discarding of empty regions. Nevertheless, this implies also that more draw calls may be necessary (even if more regions can be removed), which can decrease performance significantly.

The proposed method can be further improved by not only using the xz-plane to decide if point sprite groups can be discarded. Because many generated clouds

(especially if they are just forming up) are not so big that all slices (viewed from the top of the volume) are touched by them, there are still point sprite groups left that could be removed. To do this, more than a single density map for testing is used, which means that the slices of the volume are grouped to compute density maps for different heights (i.e. density map 1 is built by adding slice 0 to 3 together, density map 2 consists of slices 4 to 7, etc.). These maps are then downsampled as described above and used to decide if a region could be discarded.

Of course there is some extra overhead when using multiple density maps, since each of them needs to be downscaled and transfered to system memory separately. Since this has to be done only once for each simulation step, it is a simple and efficient method to decrease the number of necessary point sprites.

Note that if interpolation between volume simulation timestep t_{i-1} and t_i is used, it is also necessary to use two density maps. To decide if a segment needs to be drawn, the larger value of both downscaled density maps is used. Only if both values are zero, the area can be skipped.

7.3. Visual Quality Improvements Using Effects Based on Clouds

Even if this lies not within the scope of this thesis, the two visual quality improvement features proposed by Dobashi et al. [4] will be shortly discussed to describe how it is possible to implement them for interactive environments.

7.3.1. Ground Shadows

Clouds can usually cast shadows on the ground. To map cloud shadows to a ground mesh (which could be a plane or even a complex terrain) it is necessary to generate a shadow map [23] first. Since accuracy for ground shadows is not very important and the user would not notice small discrepancies, an inverted version of the density map introduced in Subsection 7.2.2 can be used as shadow map.

This map is then projected onto the ground depending on the light direction vector as shown in Figure 7.6.

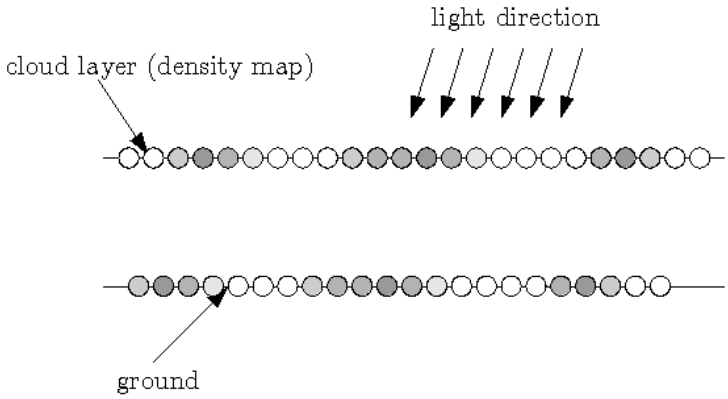


Figure 7.6: Density map is projected to ground using the light direction vector.

7.3.2. Shafts of Light

Shafts of Light can be adapted for real-time environments following the same idea Dobashi [4] described. Shells of different size which are textured using the the density map (which is introduced in Subsection 7.2.2) in a projective way, are rendered using alpha blending. Just one simplification is used compared to Dobashi's method. Instead of solving a computationally expensive integral, the effect is simply limited to areas "near the sun" by computing the 2D position of the sun on the cloud layer and then using this position to decrease the effect for long distances from this point.

8. Results

In this section, performance using the optimization methods from Section 7 will be examined and compared with methods described in Section 2.

The graphics hardware used for these performance tests is an NVIDIA GeForce 8700M GT with 512 MB GDDR3 RAM. This hardware has 32 shader units, compared to 128 units of the GeForce 8800 GPU of desktop systems, which means that it is significantly slower.

Unless stated otherwise, the test scene is rendered at a screen resolution of 1024x768 pixels. The dimensions of the cloud volume are 256x32x256, which means that more than two million voxels are processed.

Since the proposed technique consists of three (nearly) independent steps, their results will also be discussed separately as far as it is possible.

8.1. Simulation Step

8.1.1. Computation within a Single Frame

For this section, the update of the region volume will be assigned to the simulation step, since it is usually called immediately before the volume simulation step update routine.

Also, it will be considered that the method which smooths the simulation step results also belong to the simulation step (because it is usually called after the volume simulation step update routine).

Since it is necessary to perform a density map computation (Section 7) for empty space skipping, this step will also be analyzed within this section because density maps are usually generated after the volume smoothing step.

Step	Time in milliseconds	Updates per second
Region volume update	5.263	190
Simulation step update	12.5	80
Volume smoothing	6.024	166
Density map computation	0.588	1700

Table 8.1: Update rates for all parts of the simulation step.

Table 8.1 shows possible update rates per second for each part of the simulation step. All steps together can be computed about 45 times per second on a GeForce 8700M GT.

With his method based on fluid flow equations, Harris [17] has achieved about 27 simulation steps per second on an NVIDIA GeForce FX Ultra. This does not include copying of the volume data (which is stored on a 2D texture) to a 3D texture, which is also expensive due to the fact that a huge amount of memory needs to be transferred between graphics memory and system memory.

To perform a fair test, only the region volume update and the simulation step update will be considered. These two steps can be processed about 66 times per second on the above-mentioned hardware. Since Harris [17] tested his technique on a NVIDIA GeForce FX Ultra, an older graphics adapter is also used to compare performance. On an NVIDIA GeForce 6800 Ultra, the two steps in our algorithm can be processed 50 times per second using also a volume resolution of 256x32x256. Harris achieved with his method based on fluid flow equations about 27 simulation steps per second. Even if we assume the 6800 Ultra is two times faster than the FX Ultra, the method proposed by Harris would not allow more than 54 steps per second on a GeForce 6800 Ultra using a cloud volume that is 64 times smaller than the one used in the test environment – therefore, our method can be considered about one order of magnitude more efficient.

Dobashi's bitfield manipulation approach, including the smoothing step, can't be performed at interactive framerates using a test-application running on a Pentium IV CPU with a clock of 2.4 GHz.

8.1.2. Computation over Multiple Frames

Since the method is well suited to be processed over multiple frames (because

updates are not necessary for each frame), splitting the workload leads to high performance improvements because the above-mentioned steps are mainly processed within the pixel shader and therefore they are fillrate intensive.

In a test scenario (using the same environment as described above), splitting in a way that only a single slice (of 32 slices) is processed per frame, the simulation did not affect framerate of the application. The only drawback is, that changes in the cloud field are limited in speed since the simulation is distributed over several frames. Assuming the application runs at 60 frames per second and only one slice is processed each frame, this means that 128 frames are necessary (32 for region volume slice update, 32 for volume simulation update, 32 for smoothing and 32 for densitymap computation) before results of a simulation step are available. Running at 60 frames per second therefore means that simulation step updates occur only every 2.13 seconds, which will be enough for most purposes but can also be too slow for fast simulations. If faster simulations are necessary, it is also possible to process more than a single slice (i.e. 4 slices) per frame.

8.2. Shading Step

The proposed shading technique is able to compute attenuation for every single voxel about 8.696ms (also tested on a GeForce 8700M GT). Performance can be significantly increased by reducing resolution of the shading map. For example, using only the half resolution of the cloud volume map (which corresponds to a shading volume with dimensions of 128x16x128), shading computations could be performed about 460 times per second. Again, the computations can be distributed over several frames to avoid fillrate limitations of the application.

8.3. Rendering Step

Rendering performance depends mainly on the cloud cover while performance of the simulation and shading steps are independent of how many cloud particles are set. For this section, three different cases will be considered. For case one, a low cloud cover (approximately 15-20%) is used. For case two, a cloud cover of approximately 40% is used and for case three, a very dense cloud cover (over 70%) is used.

Without the improvements discussed in Section 7, rendering of the cloud field is relatively slow, as can be seen in Table 8.2. to Table 8.4.

Performance measurements without improvements		
Low cloud cover	**Medium cloud cover**	**High cloud cover**
33 fps	25 fps	11 fps

Table 8.2: Performance comparison without improvements of Section 7 enabled.

Performance measurements using VFD test		
Low cloud cover	**Medium cloud cover**	**High cloud cover**
95 fps	51 fps	21 fps

Table 8.3: Performance comparison using the viewing frustum test described in Section 7.

Performance measurements using VFD/ESS test		
Low cloud cover	**Medium cloud cover**	**High cloud cover**
118 fps	55 fps	33 fps

Table 8.4: Performance comparison using the viewing frustum test and the empty space skipping test described in Section 7.

As can be seen in Tables 8.2 to 8.4, the performance highly depends on the cloud density. Using no improvements, the rendering process is very slow and does not run at interactive framerates for a highly covered sky. A simple viewing-frustum test already improves performance significantly. Using empty space skipping as described in Section 7 further improves performance and therefore allows rendering at higher framerates.

All values are measured under identical conditions using the same scene setup.

Because simulation- and shading steps can be easily split to be computed on multiple frames, they are practically not affecting the framerate and therefore fit well into most environments, as long as they are not already fillrate limited. Rendering is currently the bottleneck of the described technique.

Even if high interactive framerates can be achieved, there is still much space for improvements. Currently, Shader Model 3.0 is necessary for cloud rendering since

vertex texture fetches (which are used to decide if a point sprite of the group needs to be drawn) are not supported on older hardware. In Section 9, a method to improve cloud rendering using Shader Model 4 is discussed in short, which would allow reducing the number of vertices to process significantly.

9. Conclusions and Future Work

In this thesis, we have presented a technique to perform realistic cloud simulation, shading and rendering which improves on previous methods by allowing real-time rendering and dynamic shading computations.

The proposed technique is based on Dobashi's paper *A Simple, Efficient Method for Realistic Animation of Clouds*, in which a very flexible approach for dynamic cloud simulation is proposed, but which is also restricted to non-interactive environments.

Since previous methods are restricted in simulation options (formation of clouds, motion, extinction), dynamic shading or rendering, the main goal of this thesis, which was defined in Section 1, was to develop a cloud rendering system that fits well to existing methods for sky rendering, allows dynamic cloud simulation, real-time shading of clouds and rendering of the results at interactive framerates.

To determine if the approach is successful, the list of quality criteria, which was introduced in Section 1 to analyze the quality of previous work is revisited.

9.1. Quality Review of the Proposed Technique

9.1.1. Dynamic Simulation

- ✔ Realistic formation of clouds
- ✔ Cloud movement
  - ✔ advection by a single wind vector
  - ✔ advection by a more complex flow field
- ✔ Cloud extinction
- ✔ Nearly stable user-defined cloud cover
- ✔ Control over cloud shapes; allow animators to define shapes which should be built during simulation

9.1.2. Dynamic Shading and Lighting

- ✔ Realistic lighting through scattering
  - ✔ single scattering, using one ray per cloud particle in combination with an ambient term instead of multiple scattering, which is computationally more expensive.
- ✔ Advection by sun- and sky color

9.1.3. Performance Issues

- ✔ Possibility to perform cloud simulation in real time
  - ✔ use GPU for nearly everything
- ✔ Possibility to perform cloud rendering in real time using point sprite groups and optimizations as empty space skipping.
  - ✔ avoid memory transfers between main memory and graphics adapter; the only necessary transfer of resources from graphics memory to main memory is the downsampled density map for empty space skipping during rendering, but this map is mostly limited to sizes of 8x8 or 16x16 pixels and does therefore not influence performance.

The list together with the quantitative analysis in the previous section proves that all main goals of the thesis have been reached.

This method is easy to use and grants a high degree of artistic control while it remains easy to generate a realistic-looking cloud volume randomly without any artist influence. Parameters such as cloud density, average cloud size, wind speed and direction can be easily integrated into the region volume computations (Section 4) and be used to give the user maximum control over the cloud field, which allows for example changes in the weather pattern. This makes the proposed method suitable for most applications, reaching from games and flight simulators to virtual environments.

9.2. Future Improvements

There is still room left for improvements. For latest graphics hardware, it would be possible to reduce the memory consumption and increase performance of the simulation step significantly by implementing the bitfield manipulation functions (as described by Dobashi et al. [4]) within the shader for execution on the GPU. This would improve performance significantly, since 32 voxels could be processed

(using 32 bit integer textures) simultaneously by simply storing each voxel within a single bit and perform computations using boolean operations.

It may be possible to further improve the cloud rendering step by using geometry shaders, which are part of Shader Model 4.0. Using these shaders, it would be possible to generate the necessary point sprites on the fly instead of having to use a vertex stream that holds for example 32x32 point sprites and where sprites need to be discarded within the vertex shader if no particle exists on the corresponding volume position.

Further improvements concerning visual quality are still open. For example, support for local light sources could be integrated to illuminate clouds by thunderbolts or even aircrafts. Especially methods to perform a fast multiple scattering approximation need to be developed. Làszlo Szirmay-Kalos proposed in *Real-Time Multiple Scattering in Participating Media with Illumination Networks* [24] a very interesting approach for multiple scattering computation at interactive framerates for static clouds. Maybe an adaption of this method for dynamically generated clouds can be found in the future.

Appendix A – Implementation

Most of the necessary computations are performed within vertex- and pixel shaders and therefore, this section will focus mainly on the shader code, since the program code itself is mainly restricted to set shader constant parameters and performing draw calls.

Cloud Simulation

The vertex shader only transforms vertices of a simple quad that is used to render to a texture. Only the interesting pixel shader parts will be listed.

```
float4 SimulationStepPS( VS_OUTPUT In) : COLOR0
{
    float4 vResult;
    //get current states at actual map position
    float4 vLastStep = tex2D(SamplerLastStep, In.TexCoord.xy);

    //compute new humidity value with hum(t+1) := hum(t) and not act(t)
    //note, that the humidity value is written to the r-component
    if((vLastStep.r >= 0.5) && (vLastStep.b < 0.5)) {
      vResult.r = 1.0;
    }
    else {
      vResult.r = 0.0;
    }

    //compute new cloud value with cld(t+1) = cld(t) or act(t)
    //note that the cld value is written to the g-component
    if((vLastStep.g >= 0.5) || (vLastStep.b >= 0.5)) {
      vResult.g = 1.0;
    }
    else {
      vResult.g = 0.0;
    }

    //compute new activation value with act(t+1) = not act(t) and hum(t) and f_act
    //where f_act is definied in the original publication by Dobashi et al (A
    //Simple, Efficient Method for Realistic Animation of Clouds).

    //first, compute the actual segment number
```

```
float2 vActSegment;
float fActSegmentIndex;
float fTempSegmentIndex;

float2 vTemp;
float2 vTemp2;

vTemp.x = In.TexCoord.x * 2047.0;
vTemp.y = In.TexCoord.y * 1023.0;

vTemp2.x = In.TexCoord.x / fSegmentWidth;
vTemp2.y = In.TexCoord.y / fSegmentHeight;

vActSegment.x = round(vTemp2.x);
vActSegment.y = round(vTemp2.y);
if((vActSegment.x - vTemp2.x) > 0.0) vActSegment.x = vActSegment.x - 1.0;
if((vActSegment.y - vTemp2.y) > 0.0) vActSegment.y = vActSegment.y - 1.0;

fActSegmentIndex = round(vActSegment.x + (vActSegment.y * 8.0));

//compute local uv-offset
float2 vLocalUV;
vLocalUV.x = In.TexCoord.x - (vActSegment.x / 2048.0 * 256.0);
vLocalUV.y = In.TexCoord.y - (vActSegment.y / 1024.0 * 256.0);

float fAct;
float fAct1 = tex2D(SamplerLastStep, float2(In.TexCoord.x +
                                            fOffsetX1, In.TexCoord.y)).b;
float fAct2 = tex2D(SamplerLastStep, float2(In.TexCoord.x, In.TexCoord.y +
                                            fOffsetZ1)).b;
float fAct3 = tex2D(SamplerLastStep, float2(In.TexCoord.x - fOffsetX1,
                                            In.TexCoord.y)).b;
float fAct4 = tex2D(SamplerLastStep, float2(In.TexCoord.x, In.TexCoord.y -
                                            fOffsetZ1)).b;
float fAct5 = tex2D(SamplerLastStep, float2(In.TexCoord.x - fOffsetX2,
                                            In.TexCoord.y)).b;
float fAct6 = tex2D(SamplerLastStep, float2(In.TexCoord.x + fOffsetX2,
                                       In.TexCoord.y)).b;
float fAct7 = tex2D(SamplerLastStep, float2(In.TexCoord.x, In.TexCoord.y -
                                       fOffsetZ2)).b;
float fAct8 = tex2D(SamplerLastStep, float2(In.TexCoord.x, In.TexCoord.y +
                                       fOffsetZ2)).b;

float fAct9 = CheckOtherSegment(fActSegmentIndex - 1.0, vLocalUV);
float fAct10 = CheckOtherSegment(fActSegmentIndex - 2.0, vLocalUV);
float fAct11 = CheckOtherSegment(fActSegmentIndex + 1.0, vLocalUV);

if((fAct1 >= 0.5) || (fAct2 >= 0.5) || (fAct3 >= 0.5) || (fAct4 >= 0.5) ||
   (fAct5 >= 0.5) || (fAct6 >= 0.5) || (fAct7 >= 0.5) || (fAct8 >= 0.5) ||
   (fAct9 >= 0.5) || (fAct10 >= 0.5) || (fAct11 >= 0.5)) {
```

```
  fAct = 1.0;
}
else {
  fAct = 0.0;
}

if((vLastStep.b < 0.5) && (vLastStep.r >= 0.5) && (fAct >= 0.5)) {
  vResult.b = 1.0;
}
else {
  vResult.b = 0.0;
}


float2 vNoise;
float fNoiseValue;

//get density scale value for slice
float fSliceDensity = tex1D(SamplerSliceDensity, fActSegmentIndex / 31.0).r;

//fetch values from regionmask textures
float fRegionMaskHumAct = tex2D(SamplerRegionVolume,
                                float2(In.TexCoord.x, In.TexCoord.y)).r;
float fRegionMaskExt = 1.0 - fRegionMaskHumAct;

//compute cloud extinction
vNoise = float2(In.TexCoord.x / 2.0 + fNoiseOffsetExtX, In.TexCoord.y / 2.0 +
                fNoiseOffsetExtY);
fNoiseValue = tex2D(SamplerNoise, vNoise).r;

if(fNoiseValue <= (fRegionMaskExt * 0.1)) {
  vResult.g = 0.0;
}

//compute humidity regeneration
vNoise = float2(In.TexCoord.x / 2.0 + fNoiseOffsetHumX, In.TexCoord.y / 2.0 +
                fNoiseOffsetHumY);
fNoiseValue = tex2D(SamplerNoise, vNoise).r;

if(fNoiseValue < (fRegionMaskHumAct * 0.1)) {
  vResult.r = 1.0;
}

//compute activation values
vNoise = float2(In.TexCoord.x / 2.0 + fNoiseOffsetActX, In.TexCoord.y / 2.0 +
                fNoiseOffsetActY);
fNoiseValue = tex2D(SamplerNoise, vNoise).r;

if(fNoiseValue < (fRegionMaskHumAct * 0.001)) {
  vResult.b = 1.0;
```

```
    }
    //remove cloud particles outside of the defined ellipsoids
       if(fRegionMaskHumAct <= 0.001) {
         vResult.g = 0.0;
       }

    //alpha channel is not used, so set it simply to zero. maybe it could be used
    //to store cloud regions in future versions.
    vResult.a = 0.0;

    return vResult;
}
```

Cloud Shading (for XZ slice orientation)

```
float4 VolumeShadingPS( VS_OUTPUT In) : COLOR0
{
  float2 vVolumePos;
  float2 vShadingPos;

  vVolumePos.x = In.TexCoord.x * fPartX + fVolBoundingMinX + fLightOffsetX;
  vVolumePos.y = In.TexCoord.y * fPartY + fVolBoundingMinY + fLightOffsetY;

  vShadingPos.x = In.TexCoord.x * fPartX + fVolBoundingMinX+fLightOffsetXShading;
  vShadingPos.y = In.TexCoord.y * fPartY + fVolBoundingMinY+fLightOffsetYShading;

  float fVolumeValue;
  float fVolumeNeighbour1;
  float fVolumeNeighbour2;
  float fVolumeNeighbour3;
  float fVolumeNeighbour4;

  if((vVolumePos.x >= fVolBoundingMinX) && (vVolumePos.x <= fVolBoundingMaxX) &&
     (vVolumePos.y >= fVolBoundingMinY) && (vVolumePos.y <= fVolBoundingMaxY)) {
    fVolumeValue = tex2D(SamplerVolume, vVolumePos).g;
  }
  else {
    //out of boudings - so there is no cloud particle on this position
    fVolumeValue = 0.0;
  }

  //left neighbour
  if(((vVolumePos.x - fPixelWidth) >= fVolBoundingMinX) && ((vVolumePos.x -
     fPixelWidth) <= fVolBoundingMaxX) && (vVolumePos.y >= fVolBoundingMinY) &&
     (vVolumePos.y <= fVolBoundingMaxY)) {
    fVolumeNeighbour1 = tex2D(SamplerVolume, float2(vVolumePos.x - fPixelWidth,
                              vVolumePos.y)).g;
```

```
}
else {
  //out of boundings - so there is no cloud particle on this position
  fVolumeNeighbour1 = 0.0;
}

//right neighbor
if(((vVolumePos.x + fPixelWidth) >= fVolBoundingMinX) && ((vVolumePos.x -
   fPixelWidth)<= fVolBoundingMaxX) && (vVolumePos.y >= fVolBoundingMinY) &&
   (vVolumePos.y <= fVolBoundingMaxY)) {
  fVolumeNeighbour2 = tex2D(SamplerVolume, float2(vVolumePos.x + fPixelWidth,
                            vVolumePos.y)).g;
}
else {
  //out of boundings - so there is no cloud particle on this position
  fVolumeNeighbour2 = 0.0;
}

//top neighbor
if((vVolumePos.x >= fVolBoundingMinX) && (vVolumePos.x <= fVolBoundingMaxX) &&
   ((vVolumePos.y - fPixelHeight) >= fVolBoundingMinY) && ((vVolumePos.y -
   fPixelHeight) <= fVolBoundingMaxY)) {
  fVolumeNeighbour3 = tex2D(SamplerVolume, float2(vVolumePos.x, vVolumePos.y -
                            fPixelHeight)).g;
}
else {
  //out of boundings - so there is no cloud particle on this position
  fVolumeNeighbour3 = 0.0;
}

//bottom neighbor
if((vVolumePos.x >= fVolBoundingMinX) && (vVolumePos.x <= fVolBoundingMaxX) &&
   ((vVolumePos.y + fPixelHeight) >= fVolBoundingMinY) && ((vVolumePos.y +
   fPixelHeight) <= fVolBoundingMaxY)) {
  fVolumeNeighbour4 = tex2D(SamplerVolume, float2(vVolumePos.x, vVolumePos.y +
                            fPixelHeight)).g;
}
else {
  //out of boundings - so there is no cloud particle on this position
  fVolumeNeighbour4 = 0.0;
}

float fSmoothedVolumeValue = (fVolumeValue + fVolumeNeighbour1 +
      fVolumeNeighbour2 + fVolumeNeighbour3 + fVolumeNeighbour4) / 5.0;

//compute attenuation for the current position on the volume slice
float fAttenuation = 1.0 - (1.0 - fAttenuationRatio) * fSmoothedVolumeValue;

//get result of previous shading step
float fPreviousValue;
```

```
  if((vShadingPos.x >= (fVolBoundingMinX + fPixelWidth)) && (vShadingPos.x <=
     (fVolBoundingMaxX - fPixelWidth)) && (vShadingPos.y >= (fVolBoundingMinY +
     fPixelHeight)) && (vShadingPos.y <= (fVolBoundingMaxY - fPixelHeight))) {
    fPreviousValue = tex2D(SamplerShading, vShadingPos).r;
  }
  else {
    fPreviousValue = 1.0;
  }

  //decrease brightness when a particle absorbed some amount of light
  float fFinalValue = fPreviousValue * fAttenuation;

  return float4(fFinalValue, fFinalValue, fFinalValue, 0.0);
}
```

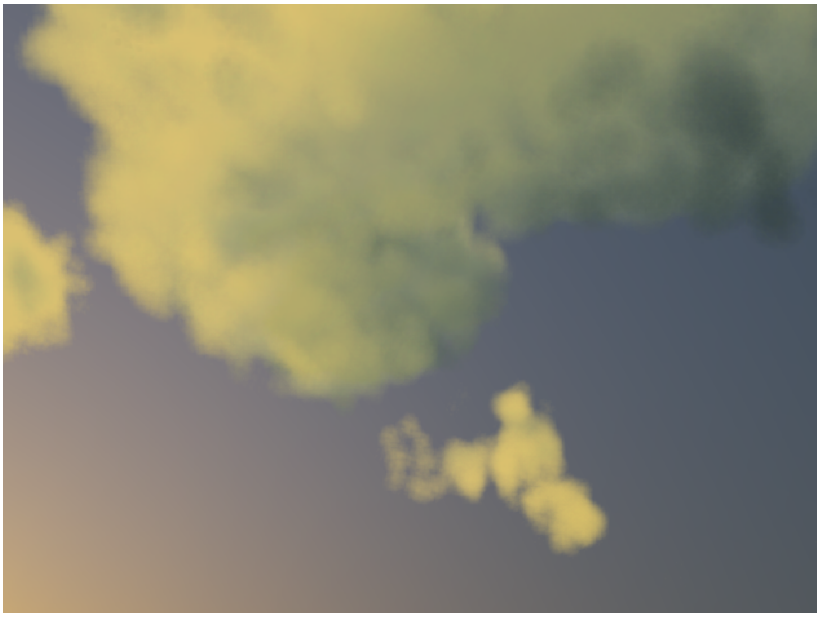
Cloud Rendering

```
VS_OUTPUT BillboardAreaVS(VS_INPUT IN)
{
    VS_OUTPUT OUT;

    float fParticle = tex2Dlod(SamplerVolume, float4(fVolumeOffsetX +
                              IN.Position.x * 0.25 * fPixelWidth, fVolumeOffsetZ +
                              IN.Position.z * 0.25 * fPixelHeight, 0.0, 1.0)).r;
    if(fParticle > 0.3) {
      float4 vCameraSpacePos = mul(float4(IN.Position.xyz + vPosition.xyz, 1.0),
                                   mtView);
      float fDistance = distance(float3(0.0, 0.0, 1.0), vCameraSpacePos.xyz);

      OUT.Position = mul(float4(IN.Position.xyz + vPosition.xyz, 1.0),
                         mtWorldViewProj);
      OUT.TexCoord = IN.TexCoord;
      OUT.Size = ((9500 * fParticle) / fDistance) * fParticle;

      OUT.Color = vSunCol * tex2Dlod(SamplerShading, float4(fVolumeOffsetX +
                                     IN.Position.x * fDistanceScale * fPixelWidth,
                                     fVolumeOffsetZ + IN.Position.z *
                                     fDistanceScale * fPixelHeight, 0.0, 1.0)).r;
      OUT.ParticleDensity = float4(fParticle, 0.0, 0.0, 0.0);
    }
    else {
```

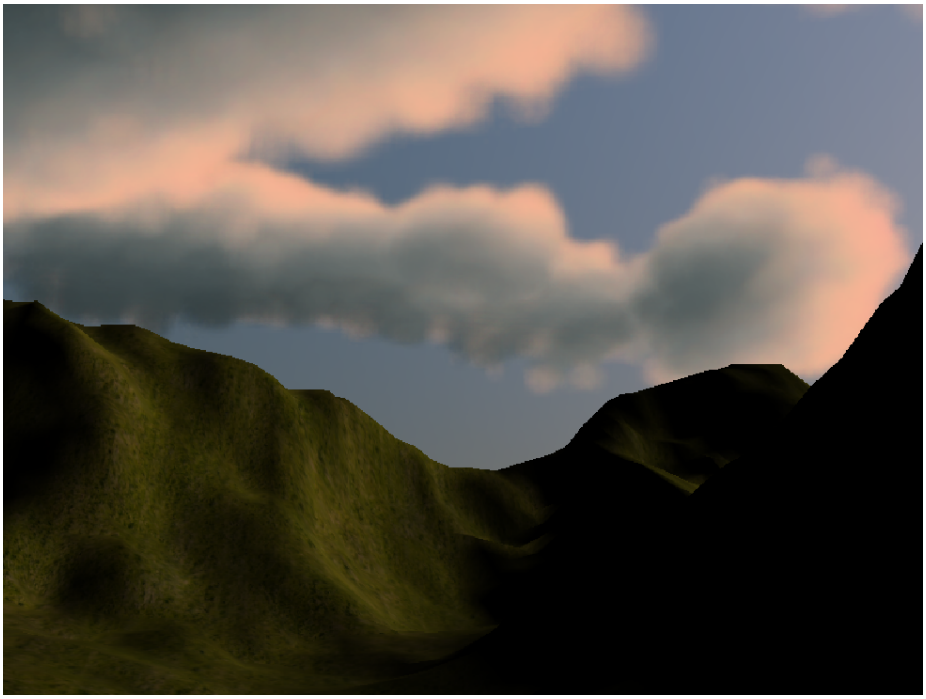
```
      OUT.Position = float4(0.0, 0.0, -1.0, 0.0);
      OUT.TexCoord = IN.TexCoord;
      OUT.Size = 1.0;
      OUT.Color = float4(0.0, 0.0, 0.0, 0.0);
      OUT.ParticleDensity = float4(0.0, 0.0, 0.0, 0.0);
    }

    return OUT;
}
```

Appendix B – Images

Cloudy sky at afternoon. The viewer is positioned at high altitude.

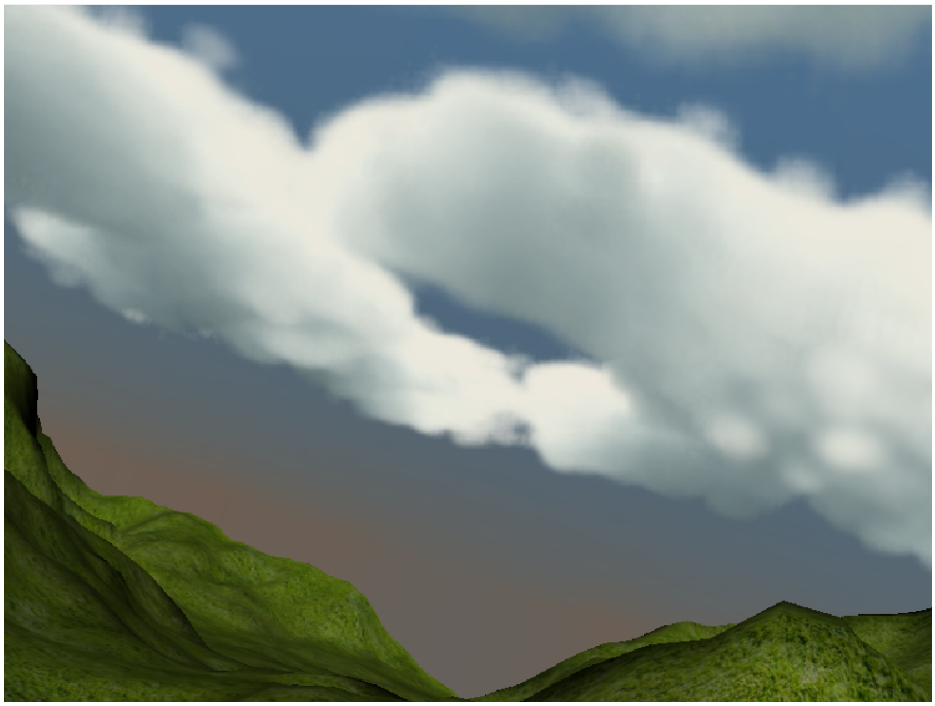


Viewer looks into the sky at sunset

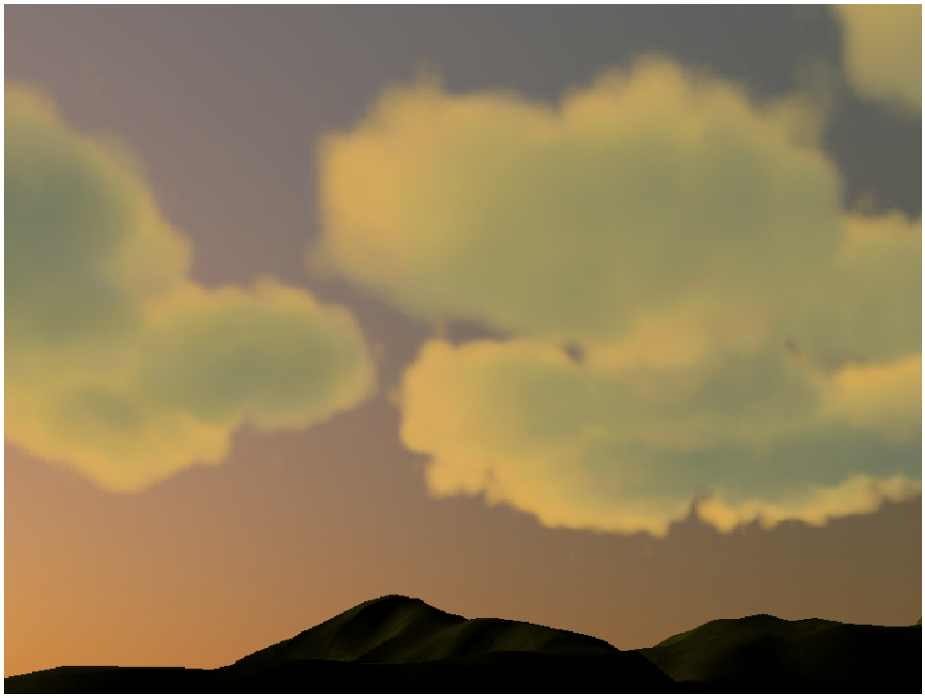


Clouds at sunset viewed from the ground.

Experimental cloud image; rendered using a cloud shading color that differs significant from the sky color to show the flexibility of the proposed method.



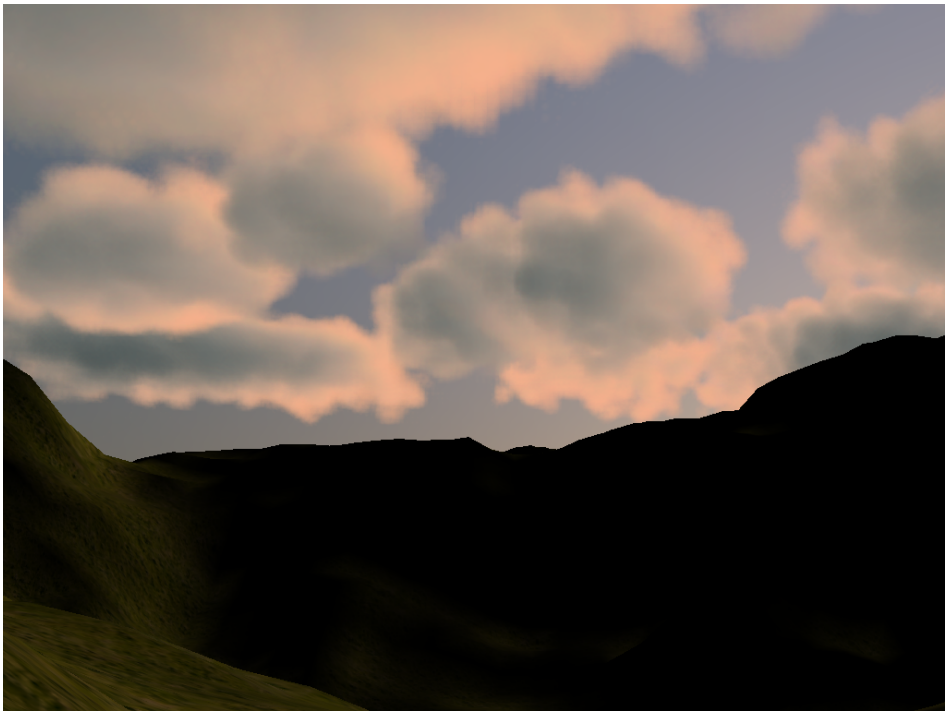
Dense clouds shown from a leaning perspective as they could be seen from an aircraft at low altitude.



Another screenshot that shows clouds at sunset.

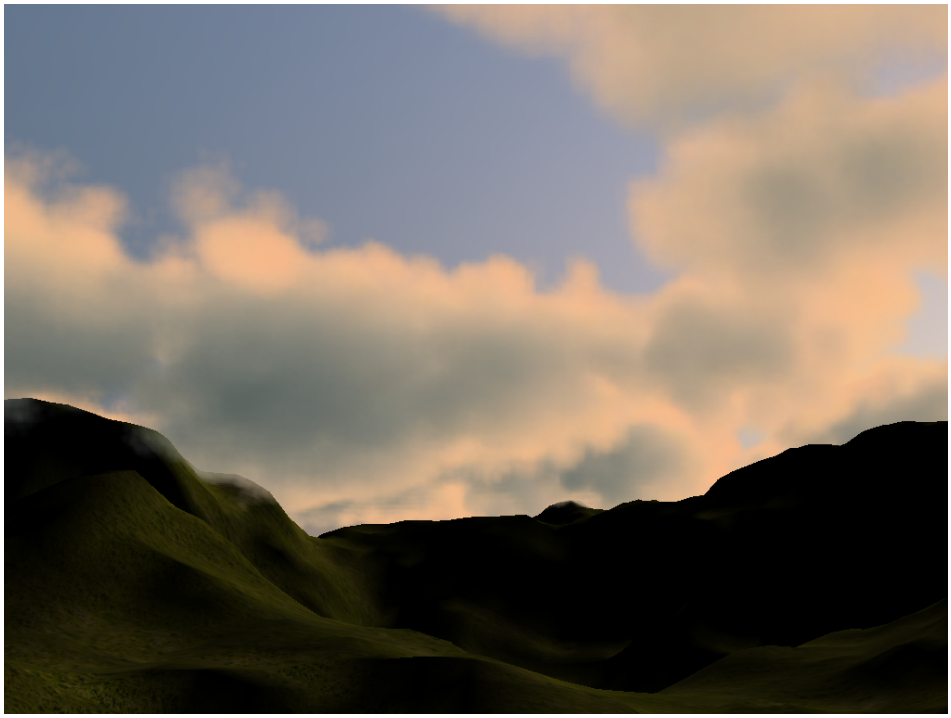
Screenshot taken on the same scene setup as for the previous shot, but using another perspective.

Viewer is positioned above the cloud field at sunset.

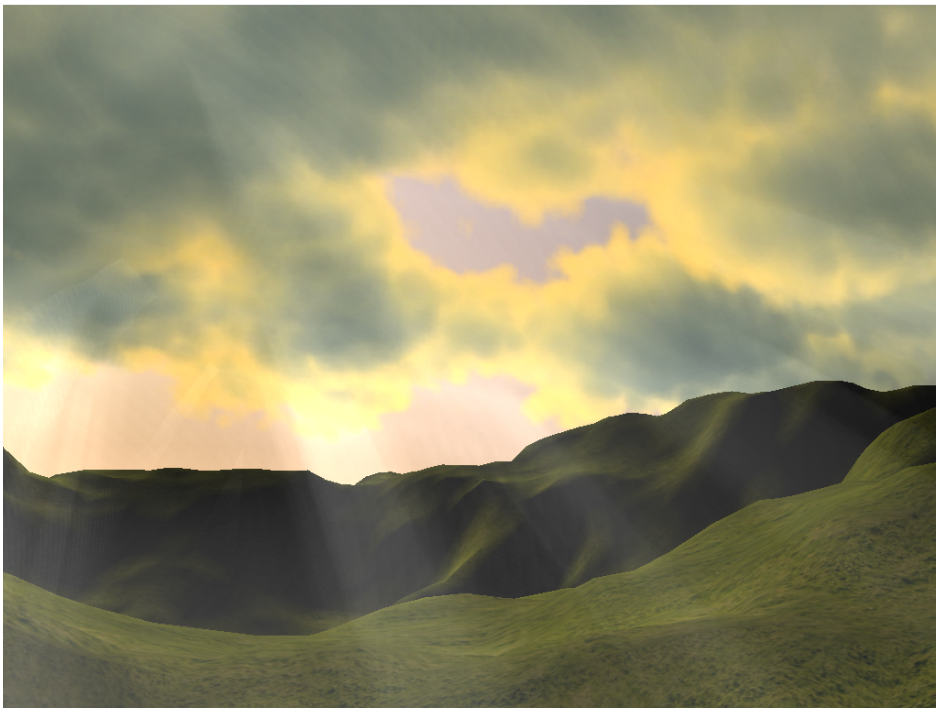


Clouds at sunset using a shading color that differs slightly from sun color.

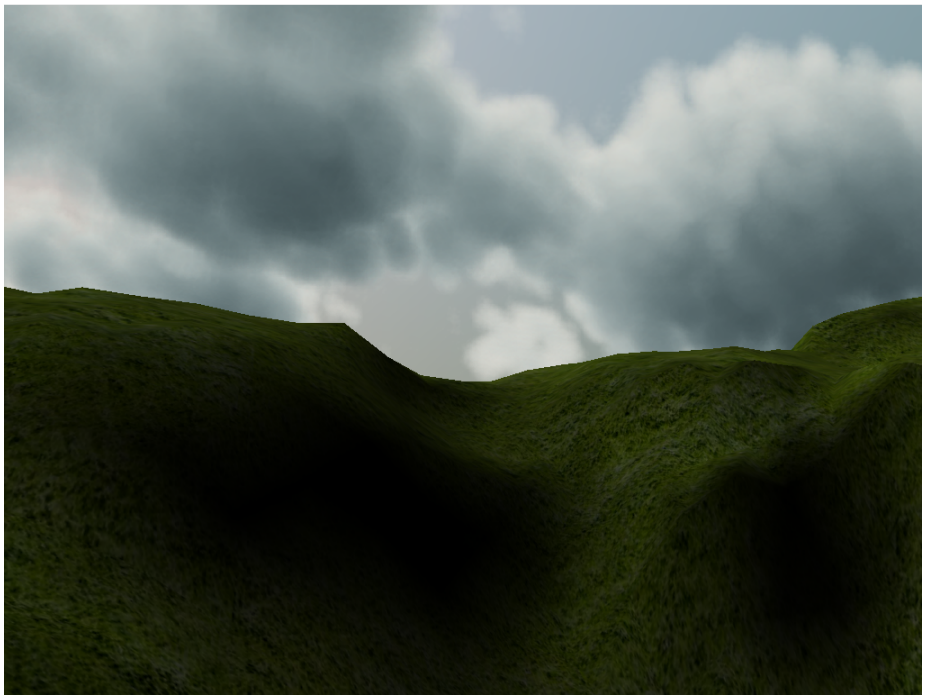
The same scene setup that was used for the previous screenshot is used to render clouds from high altitude.



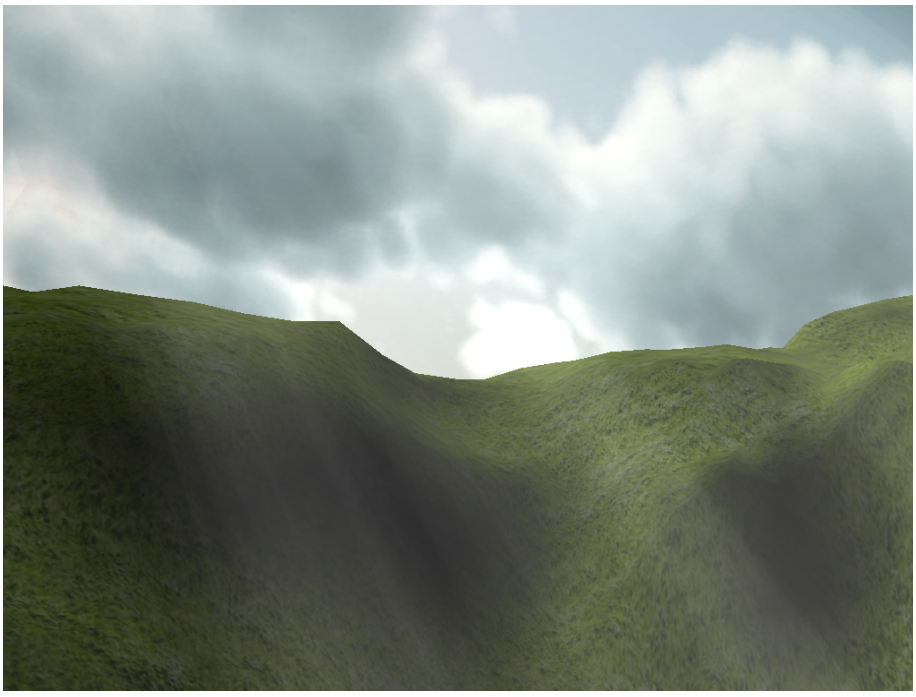
Another cloud sunset image using a slightly different shading color.



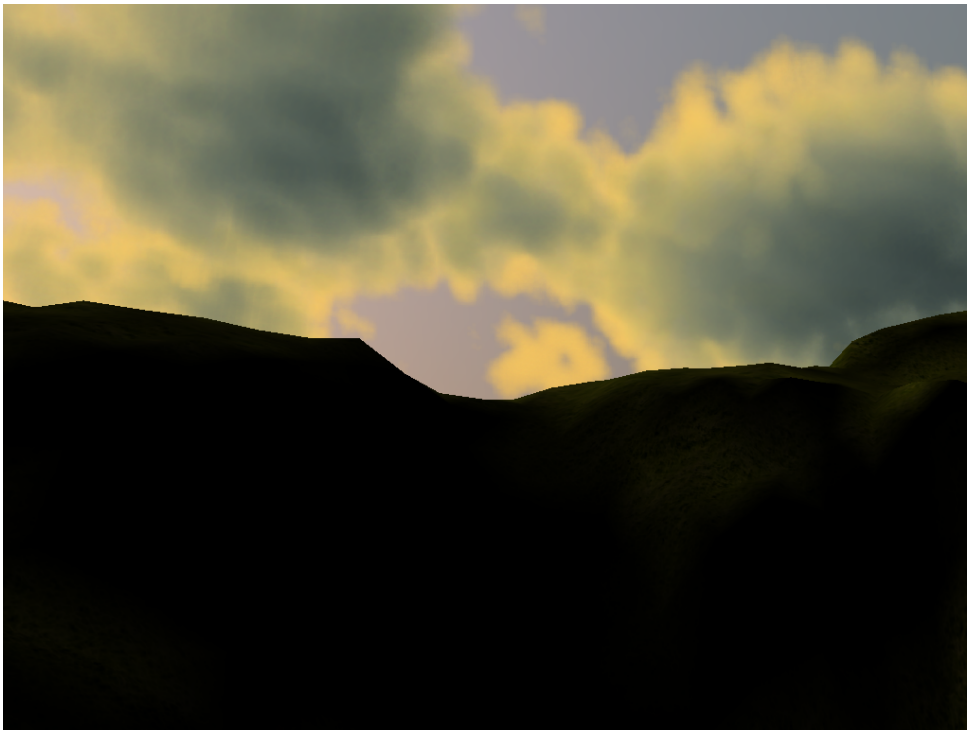
Nearly full covered sky rendered with activated shafts-of-light.



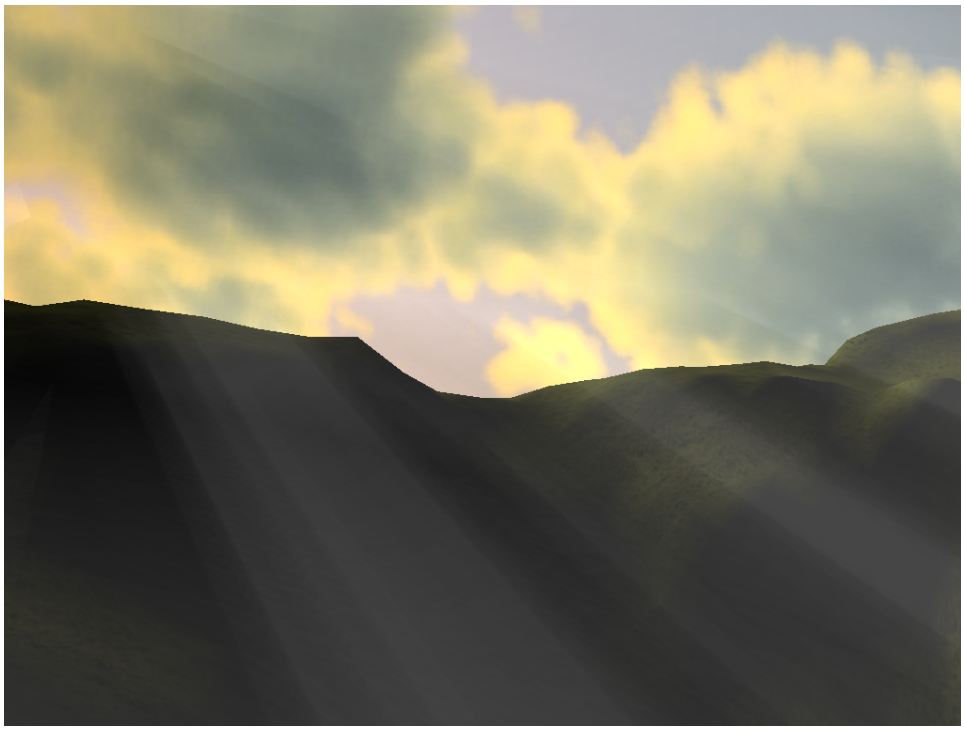
Cloud covered sky rendered without shafts-of-light.



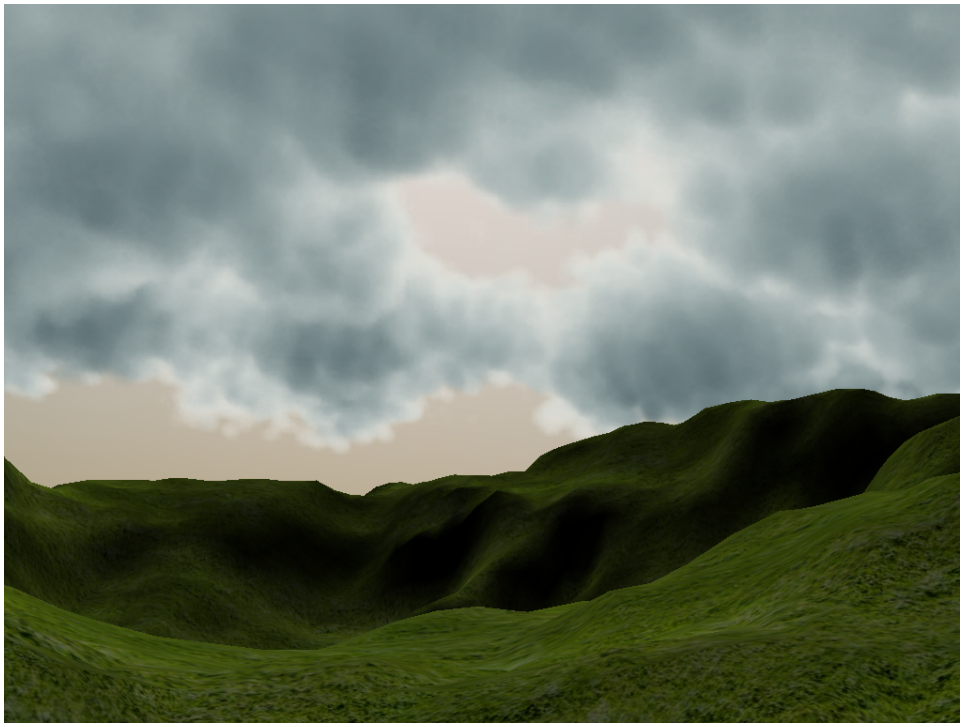
The same scene with activated shafts-of-light.



Clouds at sunset rendered without shafts-of-light.



The same scene rendered with activated shafts-of-light.

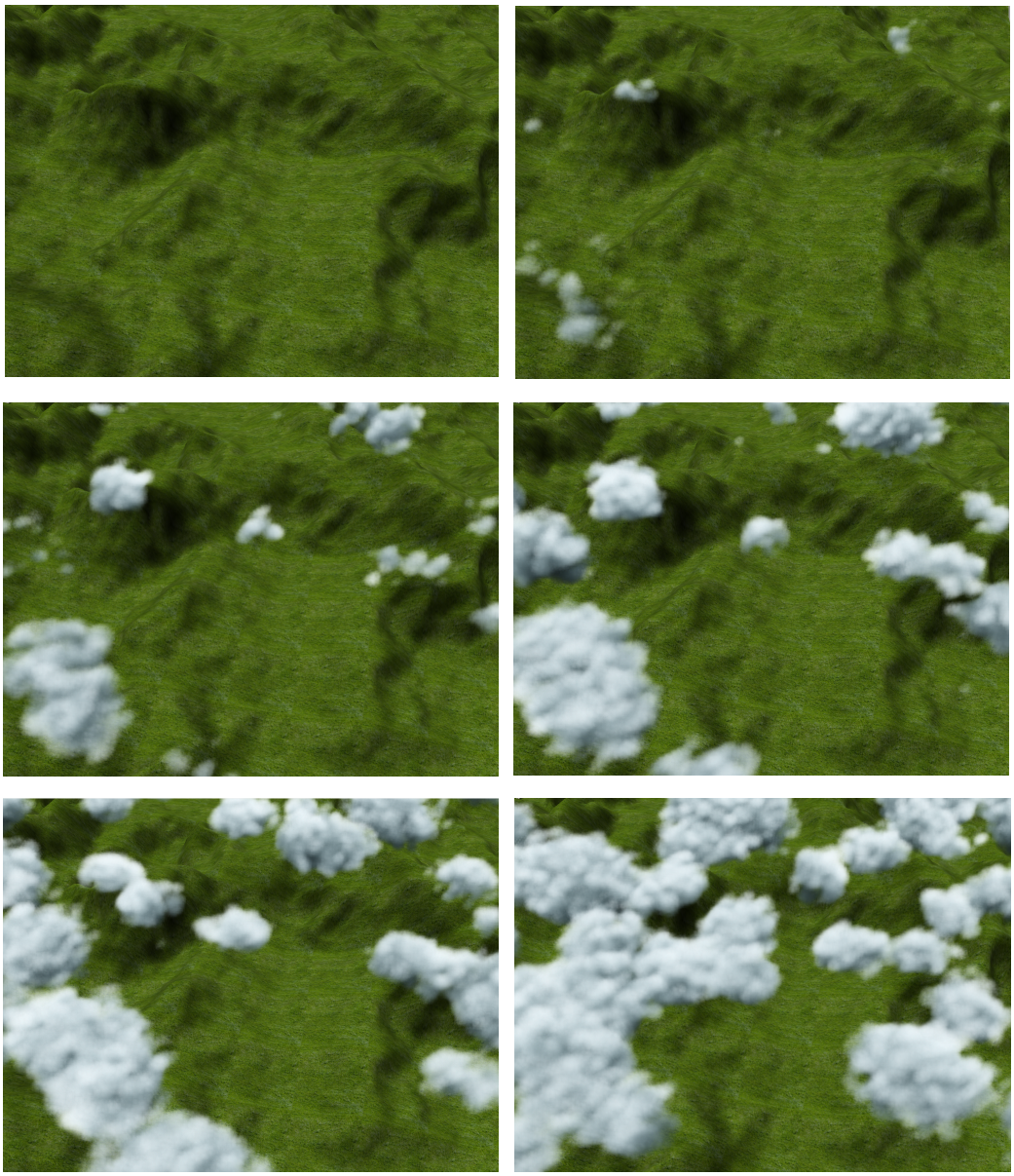


Clouds rendered without shafts-of-light.

The same scene rendered with activated shafts-of-light.

Dense clouds rendered using a low ambient value (0.1).

The same scene rendered using a higher ambient value (0.25).



The viewer is placed above the clouds looking to the ground. The images show different simulation steps during cloud formation.

List of Figures

Bibliography

[1] 3Dfx Voodoo Graphics, http://de.wikipedia.org/wiki/3dfx˙Voodoo˙Graphics

[2] A.J. Preetham, Peter Shirley, Brian Smits, "A Practical Analytic Model for Daylight", Computer Graphics, Annual Conference Series

[3] Nagel K., Raschke E., " Self-organizing criticality in cloud formation?", Institute for Theoretical Physics, W-5000 Cologne 41, Germany

[4] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, Tomoyuki Nishita, "A Simple, Efficient Method for Realistic Animation of Clouds", Hiroshima City University, Hiroshima University, University of Tokyo

[5] Yoshinori Dobashi, Tomoyuki Nishita, Tsuyoshi Okita, "Animation of clouds using cellular automation", Hiroshima City University, University of Tokyo

[6] Sim Dietrich, "Modern Graphics Engine Design", NVIDIA Corporation

[7] GPGPU - General-Purpose Computation Using Graphics Hardware, http://www.gpgpu.org

[8] Sean Whalen, Audio and the Graphics Processing Unit, 2005

[9] Sylvain Collange, Marc Daumas, David Defour, "Graphic processors to speed-up simulations for the design of high performance solar receptors", Université de Perpignan, Université Montpellier II - Sciences et Techniques du Languedoc

[10] Maria Charalambous, Pedro Trancoso, Alexandros Stamatakis, "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor", Department of Computer Science, University of Cyprus, Institute of Computer Science, Foundation for Research and Technology-Hellas, Greece

[11] Lutz Latta, "Building a Million Particle System", Massive Development GmbH

[12] NVIDIA Corporation, NVIDIA Gelato System

http://www.nvidia.de/page/gelato.html

[13] Microsoft Corporation, "DirectX SDK Documentation"

[14] Mark J. Harris, Anselmo Lastra, "Real-Time Cloud Rendering", Department of Computer Science, University of North Carolina

[15] Nelson Max, "Optical Models for Direct Volume Rendering", University of California, Davis, and Larence Livermore National Laboratory

[16] Tomoyuki Nishita, Yoshinori Dobashi, Eihachiro Nakamae, "Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light", Fukuyama University, Hiroshima University, Hiroshima Prefectural University

[17] Mark J. Harris, William V. Baxter III, Thorsten Scheuermann, Anselmo Lastra, "Simulation of Cloud Dynamics on Graphics Hardware", Department of Computer Science, University of North Carolina

[18] Ken Perlin, "An Image Synthesizer", New York University

[19] Kim Pallister, "Generating Procedural Clouds in real time on 3D HW", Intel Corporation

[20] Jean-Francois Dubé, "Realistic Cloud Rendering on Modern GPUs", Game Programming Gems 5, UBISOFT

[21] Philippe Lacroute, Marc Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation", Stanford University

[22] David Blythe, Advanced Graphics Programming Using OpenGL, 1998

[23] Cass Everitt, Ashu Rege, Cem Cebenoyan, "Hardware Shadow Mapping", NVIDIA Corporation

[24] Làszlo Szirmay-Kalos, Mateu Sbert, Tamàs Umenhoffer, "Real-Time Multiple Scattering in Participating Media with Ilumination Networks", Budapest University of Technology