VOLUMETRIC CLOUD GENERATION USING A CHINESE BRUSH CALLIGRAPHY STYLE

by
Chen Wei

Thesis Presented for the Degree of
DOCTOR OF PHILOSOPHY
in the Department of Computer Science
UNIVERSITY OF CAPE TOWN

November 2014

Supervised
by
A/Prof. James Gain
A/Prof. Patrick Marais

Acknowledgements

First, I would like to express my deepest appreciation to my supervisors Associate Professors James Gain and Patrick Marais, for their valuable suggestions and constant support while I was studying for my Ph.D. Not only did they assist me in my research but they also gave me invaluable life guidance. I am most grateful to Ingrid Gain for her continuous encouragement during some very hard times. I could not have imagined having a better team to supervise my Ph.D.

I would also like to thank the University of Cape Town and the Computer Science Department for their financial support. My sincere thanks also goes to Pierre Bezuidenhout from City Varsity for his enormous support in the user evaluation experiment.

I would like to express my sincere gratitude to my faithful friends Dr. James Lane, John Zhang, Mary Ma, Tian Zhang, Linda Zou, David Yutar, Franky Zhou, Michelle Zhang and Marina Jin for their selfless support throughout my studies.

Finally, I would like to acknowledge my dearest parents, with a love that no words can adequately express and with my deep gratitude for their sacrificial support.

Abstract

Volumetric Cloud Generation Using A Chinese Brush Calligraphy Style

Chen Wei

Supervised by A/Prof. James Gain and A/Prof. Patrick Marais

February, 2014

Clouds are an important feature of any real or simulated environment in which the sky is visible. Their amorphous, ever-changing and illuminated features make the sky vivid and beautiful. However, these features increase both the complexity of real time rendering and modelling. It is difficult to design and build volumetric clouds in an easy and intuitive way, particularly if the interface is intended for artists rather than programmers. We propose a novel modelling system motivated by an ancient painting style, *Chinese Landscape Painting*, to address this problem. With the use of only one brush and one colour, an artist can paint a vivid and detailed landscape efficiently.

In this research, we develop three emulations of a Chinese brush: a skeleton-based brush, a 2D texture footprint and a dynamic 3D footprint, all driven by the motion and pressure of a stylus pen. We propose a hybrid mapping to generate both the body and surface of volumetric clouds from the brush footprints. Our interface integrates these components along with 3D canvas control and GPU-based volumetric rendering into an interactive cloud modelling system.

Our cloud modelling system is able to create various types of clouds occurring in nature. User tests indicate that our brush calligraphy approach is preferred to conventional volumetric cloud modelling and that it produces convincing 3D cloud formations in an intuitive and interactive fashion. While traditional modelling systems focus on surface generation of 3D objects, our brush calligraphy technique constructs the interior structure. This forms the basis of a new modelling style for objects with amorphous shape.

Contents

List of Tables

List of Figures

I am the daughter of Earth and Water,
And the nursling of the Sky;
I pass through the pores of the ocean and shores;
I change, but I cannot die.

Percy Bysshe Shelley

Chapter 1

Introduction

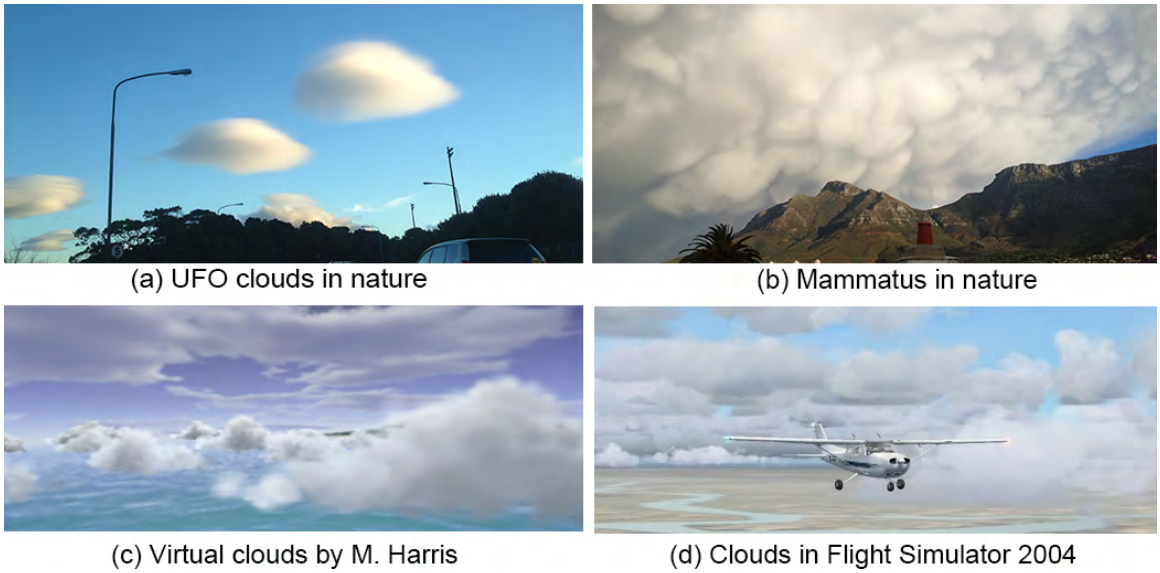


Figure 1.1: Examples of clouds in various applications.

Clouds are an important feature of any real or simulated environment in which the sky is visible as illustrated in Figure 1.1. These amorphous and subtly illuminated features serve to contribute both realism and atmosphere to a 3D scene. In the animation and game industries, a specific cloud shape or formation is commonly required by artists. However, unlike general 3D objects, which can be created accurately through surface construction methods, it remains difficult for artists to model and

render 3D clouds quickly and intuitively.
This research focuses on the creation of such complex cloud shapes from an artist's perspective. The goal is to provide an interface which artists can use to build volumetric clouds in an intuitive way, while the beauty of its shape and formation can be immediately visualized.

1.1 Overview and Motivation

To date, a number of methods have been proposed for cloud generation, primarily using procedural methods [32] or physics-based simulation [48]. Typically, these approaches provide a realistic appearance but limited shape control. Therefore, a number of hybrid methods are proposed to address the issue of shape management. These methods manipulate various proxy geometries to form the cloud shapes and generate the details of the cloud surface using procedural methods. Recently, several sketch-based techniques have been introduced to define the silhouette of cumulus clouds. These techniques provide some measure of manual control over cloud shape. Unfortunately, the required level of manual control can be painstaking and tiring for the artist. In addition, the computational times required for these methods range from several seconds to minutes, which prohibits interactive use and affects the user experience.

Figure 1.2: 'Mi Imitation' by Chun Chen (AD 1438–1544), a Chinese Mountain and River painting stored in the Palace Museum, Beijing. Note the intricate clouds reflecting elegant brush use combined with ink dispersion.

In this thesis, we were inspired by an ancient Eastern landscape painting style — *Mountain and River (MR) painting*. Figure 1.2 shows a representative painting in this style. There are some unique features that stand out in this style of painting. First, apart from mountains, rivers and trees, many MR paintings have significant cloud coverage. Second, MR paintings are more impressionistic than realistic, focusing on a spiritual element derived from the position and flow of mountains and clouds. These elements require precise and varied brush control. In particular, using only a single brush and colour, an artist can paint a varied and detailed landscape.

Figure 1.3: (a) shows a 3D cloud in the shape of a Chinese character 'home' with a hairy effect on the end of some strokes. (b) A duck shape reflects a volumetric illuminance effect with the light source behind the head of the duck. (c) A perspective view of large scale clouds in a complex formation. Note these 3D clouds are generated in less than 5 mins by the users in total. The generation of 3D clouds using our *Hybrid Mapping Method (HMM)* takes less than a second after each stroke. This provides an intuitive modelling method since artists are able to check the 3D result almost instantly.

However, it is not desirable to exactly reproduce MR painting using high fidelity simulation. From a computational perspective, this is unlikely to allow a truly responsive system when combined with the demands of volumetric cloud rendering. More importantly, an MR painter is constrained by the medium to use negative space, defining clouds indirectly by painting around their borders and this takes considerable practice. Instead, we abstract the effects of brush tilt, velocity and pressure and the ink dispersion that results from porous rice paper to allow an expressive economy in cloud modelling. This does mean that mouse input is insufficient and our system requires a stylus pen or similar input device. Figure 1.3 presents an example of 3D clouds generated using our system.

1.2 Clouds in Nature

Clouds are so commonplace that their beauty and importance are often overlooked. Human activities throughout civilized history, have been significantly affected by weather conditions. For civilizations from Babylon to ancient China and India, clouds served as a means of short term weather prediction [100]. For instance, a morning with wispy cirrus clouds hovering in the blue sky usually indicates a day of fair weather while dark clouds with a thick volume suggests the approach of a heavy storm. Once mammatus clouds appear, Figure 1.1(b), we know the storm is moving away.

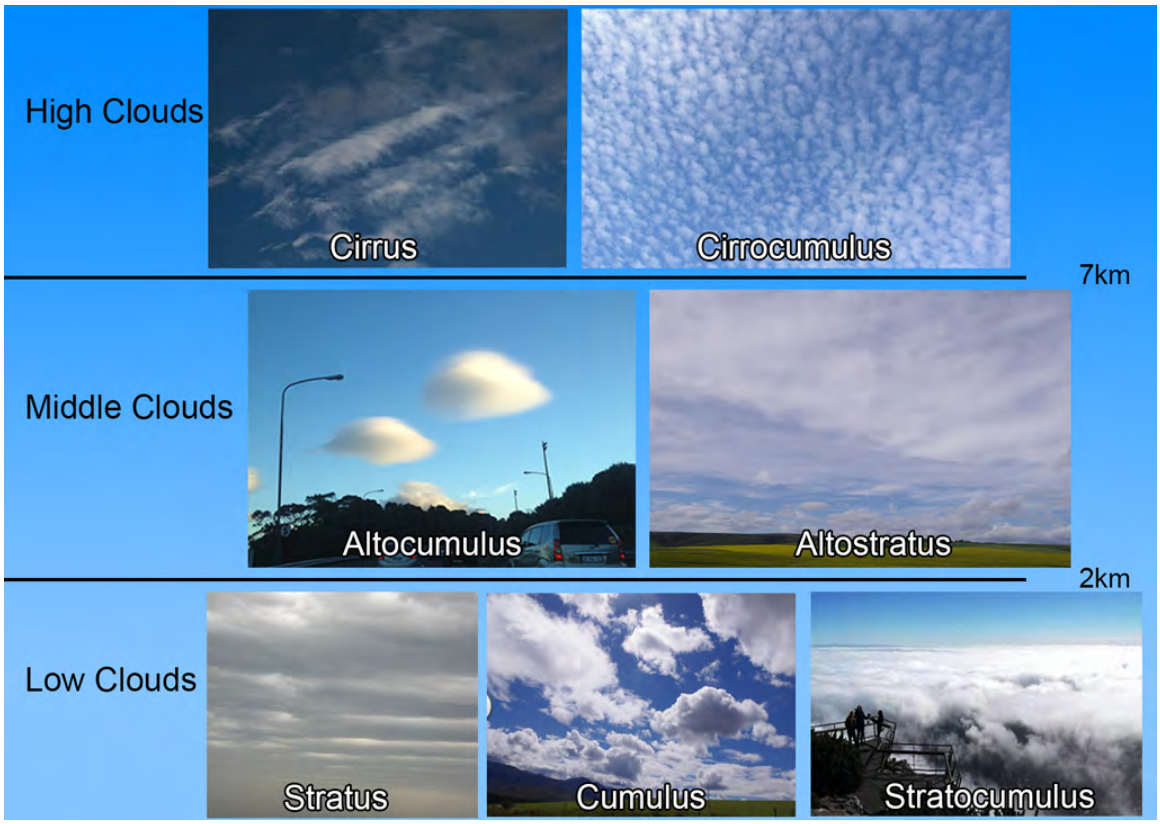


Figure 1.4: Cloud types based on altitude. Note that various cloud shapes in the same altitude.

On 1st August 1861, Admiral FitzRoy gave the first modern weather forecast with the help of the weather observation networks across countries and the telegraph, which led meteorology study into the modern era. Nowadays, meteorologists study the atmosphere through satellite images, in which clouds are one of the major research

subjects. Liquid droplets and suspended particles are the main components that form clouds in nature [116]. However, the same components can produce various cloud shapes due to the distinct temperatures and air flows on different altitudes. Thus, the altitude becomes the main criterion for clouds categorization.

Figure 1.4 illustrates the altitude-based structure of some major cloud types in nature. For clouds below 2km, the temperature is relatively warmer and the atmospheric activity is more vigorous compare to higher altitudes. Clouds at low altitude can also more easily to form a larger volume due to the fast accumulation of water droplets from near-surface evaporation. Altocumulus and altostratus are middle clouds that are located between 2 to 7 km. They are in a mixture of both water droplets and ice crystals, which leads to a brighter and less fragmented appearance. Clouds at high altitude are made of ice crystals. They usually appear in a thin and wispy, such as cirrus clouds. Besides these, some types of clouds such as cumulonimbus and nimbostratus, are developed across the three altitudes. Due to the distinct temperature, air pressure, wind flow and other elements in these three altitudes, clouds are described with various body formations and edge effects.

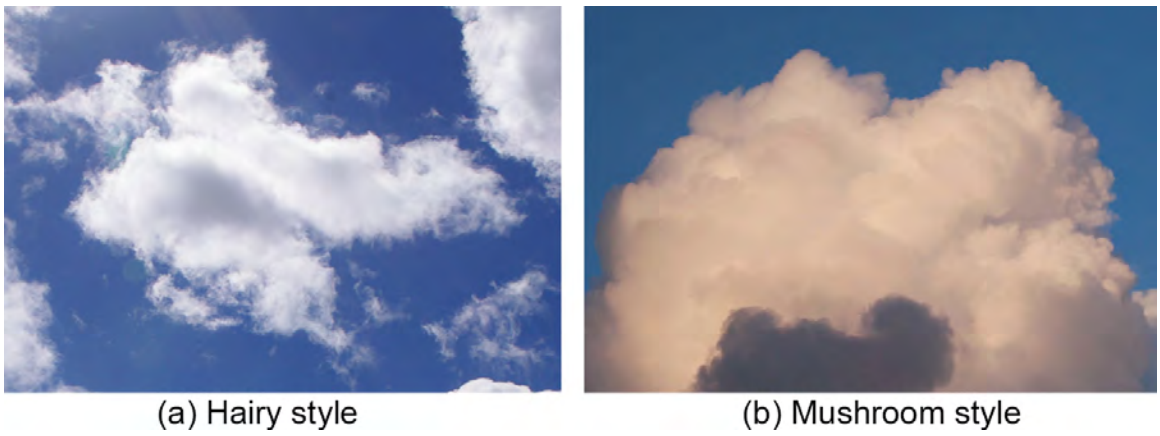


Figure 1.5: Various edge effects of clouds. (a) Hairy style with fractal patterns on the edge. (b) Mushroom style with a smooth surface.

Figure 1.5 illustrates two edge styles of clouds in nature. Their surfaces are sophisticated and ever changing. It is difficult to define their exact boundaries. Beyond their study from a scientific perspective, clouds hold a particular fascination for many poets and artists across the world. Artists paint them to express beauty as they are

considered as the poetry of the nature. In virtual environments, such as games, clouds are an important feature as their amorphous and subtly illuminated features serve to contribute both realism and atmosphere to a 3D scene.
Our research focuses on cloud modelling from the perspective of artists, particularly with a eastern painting style. For instance, the *Reversal Painting Method(Negative Space)* and *Dark Strokes on Cloud Silhouettes* are the prevailing techniques used in cloud painting, which will be elaborated on in Section 2.2.1. Artists are able to paint various types of clouds efficiently and intuitively using these methods. However, it requires the artist to have a deep understanding of how to work with a brush and canvas. It also involves fine muscle control that requires years of practise.
Thus, our goal is to leverage the brush features for cloud generation and make the process simple to use and easy to learn for 3D digital artists. Rather than simulating physical generation, the appearance of the clouds is our major concern. Cirrus, cumulus and stratus are three distinct cloud types from a visual perspective. They are different not only in scale and formation, but are also differentiated by various edge details.

1. **Cirrus** Thin with wispy edges.
2. **Cumulus** Various thickness and formation. Both the fluffy edge and smooth surface co-exist.
3. **Stratus** Large scale with smooth and plain surface.

In nature, despite of the altitude difference, from a visual perspective most clouds are a combination of these three types of clouds. Therefore, our priority is to model the cirrus, cumulus and stratus efficiently and intuitively.

1.3 Brush Simulation

Brush and ink dispersion effects are the core features of Chinese Brush Calligraphy. In our case, we simulate these features at an acceptable performance cost. Using a stylus pen, we explore three simulations of a Chinese brush:

1. **A Skeleton Brush** with a physically simulated skeleton

2. **A 2D Brush** with a static 2D sampling texture
3. **A 3D Brush** with a dynamic 3D sampling texture

These brushes produce various footprint patterns expressed through the texture samples, which are capable of replicating cloud types ranging from cumulus to nimbus.
A pilot investigation indicated that the skeleton brush is difficult for artists to manipulate. In particular, the use of the brush we simulate, is significantly different from the stylus pen, which is the final medium we work with. Thus, we subsequently developed a 2D and 3D brush, which incorporates texture-based methods. These two methods provide an easy to understand mechanism for the artists while preserving the core features of brush calligraphy. User evaluation showed that both the 2D and 3D brushes are easy to use and quick to learn for most of the users.

1.4 Cloud Generation

Once the user finishes the painting of each stroke, the 2D footprint is converted to the volumetric cloud dataset. We exploit two approaches to achieve this, a *Ray-marching Mapping Method (RMM)* and a *Hybrid Mapping Method (HMM)*.
We implemented our *Ray-marching Mapping Method (RMM)* as an initial attempt at cloud generation. It maps an arbitrary 2D footprint directly to a 3D canvas using a ray marching strategy. Although it provides real-time cloud generation performance, the generated 3D cloud displays a strong visual regularity which greatly reduces the realism.
The deficiencies of the RPM method led us to propose another approach of generating 3D clouds: the *Hybrid Mapping Method (HMM)*. This method processes the body and surface of 3D clouds differently. It preserves most features from the brush work and provides natural shapes with irregular surface details.
We implemented *HMM* on both CPU and GPU to optimize performance. Cloud generation with *HMM* takes an average of less than a second on a middle-of-the-range laptop. This significantly improves the user experience by enabling interactive design.

1.5 Volumetric Rendering

Many modelling packages use mesh-based rendering methods. Unfortunately, a cloud has an amorphous boundary structure, which is difficult to triangulate, particularly for large scale clouds such as stratus and cumulonimbus.

Our system adapts the GPU-based volumetric rendering methods of Kruger and Westermann [62] and Crane *et al.* [18]. This ray-marching approach provides immediate visualisation of the generated 3D cloud.

Furthermore, we propose two extra functions to improve the visualisation. First, we implement a GPU-based octree generation and traversal method to increase the rendering performance for sparse cloud data. Second, we incorporate volumetric lighting using a Compute Shader to improve cloud realism.

On a laptop with an i7-3610m cpu and GT650M graphics card, our volumetric rendering engine is able to visualize a 256^3 dataset in a 1280x768 window at $20 \sim 30$ fps. During the user experiment, modellers found the rendering performance satisfactory and the presented 3D clouds highly convincing.

1.6 Interface Modes

Apart from the core functional implementations of the brush simulation, cloud generation and volumetric rendering, our interface design integrates these functions into a cohesive whole. This is crucial for the success of the final user evaluation as a poor integration can affect the user experience in using the brush calligraphy method, which is the main subject of our work.

We introduce four core modes to help the artists understand the flow of our system. They are *'Camera Mode'*, *'Canvas Mode'*, *'Paint Mode'* and *'Erase Mode'*. We also implement several important functions to improve the user experience.

1. **Light Mode** to facilitate placement of the scene light
2. **A Menu System** which can be accessed directly through the stylus pen to facilitate command execution

3. **History Tracking** which significantly enhances the user experience by allowing support for an "undo" operation.

1.7 User Evaluation

We conducted an evaluation of our system with 17 users who had prior artistic and modelling experience. They were tasked with modelling a number of cloud formations to match a set of shapes. After 10 minutes practice, they performed the tasks. They reported a significant preference for our approach over alternatives, as well as rating the realism, interactivity, ease of use and ease of learning highly. Our system is regarded as providing an intuitive means of modelling 3D clouds and also generates satisfactory results for cloud visualisation.

1.8 Thesis Statement

Our hypothesis is that:

A variety of visually compelling 3D clouds can be effectively and intuitively modelled by 3D artists using a Chinese brush calligraphy style.

The following research questions arose in the course of our interface development.

*RQ*1: *Is the Chinese brush calligraphy style helpful in cloud generation?*
This is the primary question of this thesis. It requires us to study and abstract the features of Chinese Brush Calligraphy. We simulate these using a stylus pen device.

*RQ*2: *How can one simulate the Chinese brush at a low performance cost using a stylus pen?*
To answer this question, we implemented three approaches to simulate the brush use. User evaluations are conducted to evaluate which brush style is most easy to use and quick to learn.

*RQ*3: *How can one generate the desired volumetric clouds in a 3D scene based on the 2D footprint painted by the artists?*

We exploit two approaches to generate the 3D cloud data, a *Ray-marching Mapping Method (RMM)* and a *Hybrid Mapping Method (HMM)*. The goal is to produce realistic cloud shapes in a time that is acceptable for the artists.

*RQ*4: *Will the performance of volumetric rendering be fast enough for the support of cloud modelling in real-time?*

Volumetric rendering is a computational extensive technique. Based on the existing GPU supported ray-marching methods, we propose a GPU-based octree generation and traversal technique to improve the rendering performance particularly for sparse cloud data. With the help of volumetric rendering and the new interface design, the final system raises an important question:

*RQ*5: *Is our cloud modelling system intuitive to use?*

This intuitiveness is assessed through the following:

a) *Does our system produce realistic cloud shape?*

b) *Does our system present a convincing visualisation of clouds?*

c) *Is our system easy to use?*

d) *Is our system quick to learn?*

e) *Are modellers happy with our system?*

To the best of our knowledge, this work is the first to implement brush calligraphy for cloud modelling in an interactive fashion. Therefore, we employ a standard prototyping methodology in this research: outlining the problem, conducting a literature review, proposing a solution by building a prototype and finally evaluating the solution through this prototype. This prototype is a 3D cloud modelling system using a Chinese brush calligraphy style. Figure 1.6 shows the structure of our modelling system, which has four core contributions and attendant novelties: brush simulation, cloud generation, volumetric rendering and a novel interface:

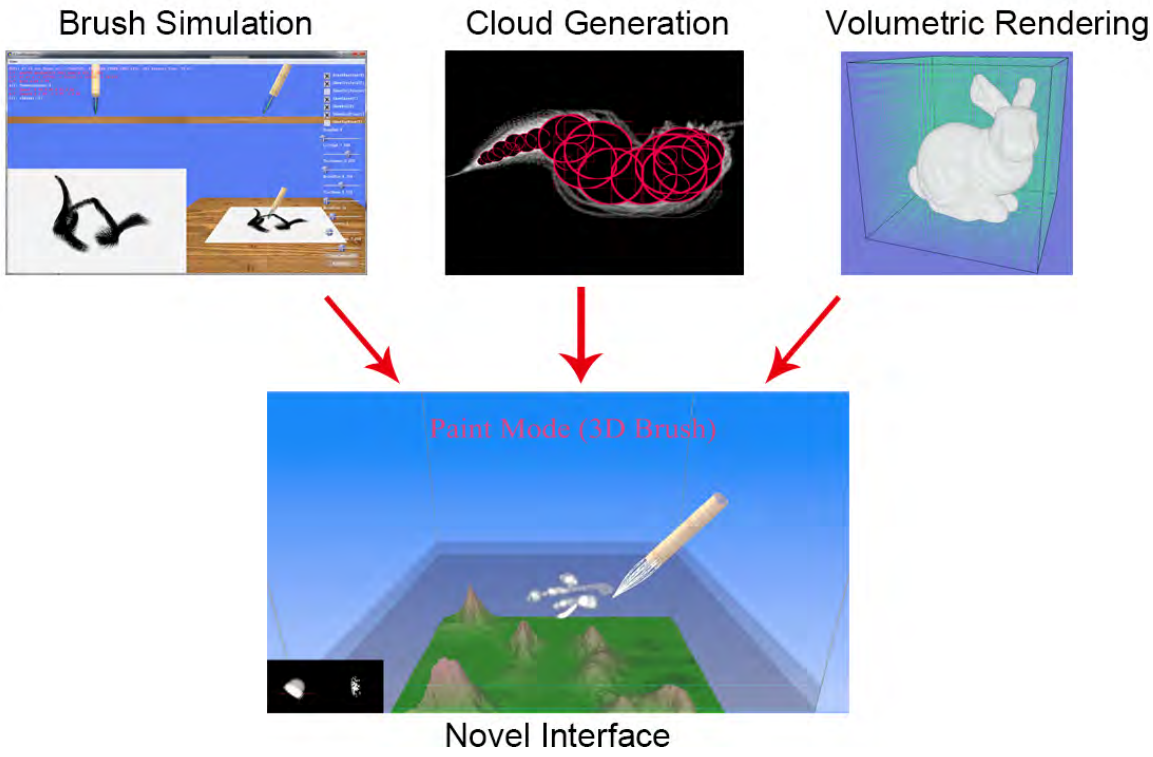


Figure 1.6: The structure of our cloud modelling system. A dynamic 3D brush is simulated to generate a 2D footprint on a 3D canvas (dark blue region). The footprint is converted to a 3D volume, which is rendered using volumetric ray-marching. These core modules are integrated within a new interface.

1.9 Contributions

This research advances the state of the art in interface techniques of 3D cloud modelling through the contributions in the following fields:

1. *Modelling Techniques*
   This work primarily proposes a novel modelling technique for volumetric cloud generation. This technique mimics the use of a Chinese brush to produce the 3D cloud in an artistic painting fashion. This makes the process of modelling straightforward and quick to learn.

2. *Brush Calligraphy Simulation*
   This work develops three methods of brush simulations, a skeleton-based brush,

a 2D brush and 3D brush. These methods cater for emulating the features of brush calligraphy at an acceptable fidelity by using a stylus pen as an input device. The results indicate that the 2D and 3D brush are able to simulate most features of brush work. This enables our novel modelling system to produce varied types of clouds such as cirrus, stratus, cumulus, and even large and complex clouds such as cumulonimbus.

3. *Texture Processing*

   We propose a novel texture process, the *Hybrid Mapping Method*, which converts an arbitrary 2D footprint to a 3D cloud dataset. This method enables realistic 3D cloud generation at interactive rates. It not only realizes a core function we require, but also presents an approach to use both the CPU and GPU to generate complex amorphous objects effectively.

4. *Volumetric Rendering*

   Volumetric rendering is a computationally costly method even using the power of the GPU. We introduce a GPU-based Octree generation and traversal method to speed up performance. In order to provide a better visualisation of 3D clouds, we also develop a fast volumetric illumination technique for light-scattering and soft-shadow effects.

5. *User Interface*

   We design a novel user interface for cloud modelling system. This interface follows the habit of traditional 2D painting by artists, which allows an intuitive use of all the integrated new techniques.

1.10 Thesis Organization

Chapter 2 provides a literature study of related work in cloud modelling and brush calligraphy simulations. This chapter identifies the niche our modelling system occupies in relation to other modelling techniques. It also provides a brief introduction to Chinese brush use through the appreciation of calligraphy and Chinese landscape painting. Core features are identified, which lead to the motivation of using it for our cloud modelling.

The next four chapters introduce the four core modules of our modelling system. Chapter 3 describes the new user interface for the modelling system; Chapter 4 introduces the three methods of Chinese brush simulations; Chapter 5 details the cloud generation and Chapter 6 presents the volumetric rendering method.

Chapter 7 presents the user evaluation of our system. Chapter 8 draws conclusions and includes recommendations for the future advancement of this research.

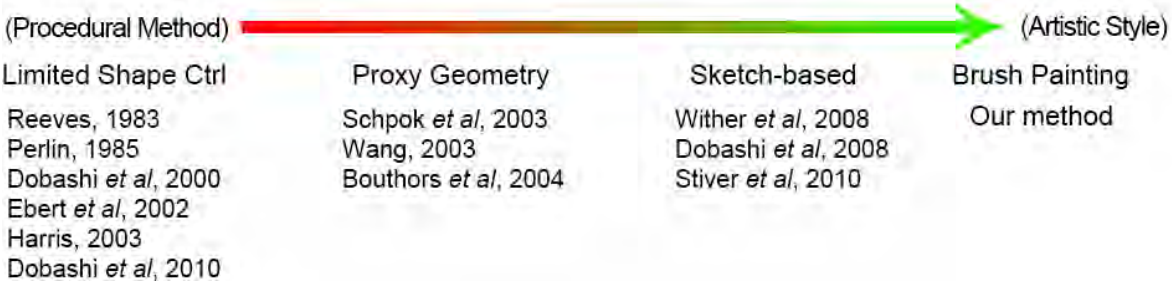
Chapter 2

Background

The brush calligraphy cloud modelling system we propose integrates a number of techniques from computer graphics and human computer interaction. In this chapter, we explain the related work of two core modules of our modelling system: cloud modelling and rendering; and brush simulation.

2.1 Cloud Modelling and Rendering

Table 2.1: A representative selection of previous work with respect to cloud modelling. This spectrum denotes the intuitiveness of modelling interface. The procedural methods have very limited shape control method during the cloud modelling, while the sketch-based methods can easily define the silhouette of a cumulus cloud. Our brush calligraphy method aims to provide an artistic painting style, which can generate the volumetric clouds in an efficient and intuitive manner.



Normally, modelling systems consist of two important components: the modelling interface and a rendering system. Distinct object representations are associated

with specific modelling and rendering techniques. For instance, Polygon objects are commonly created through surface construction methods and rendered through rasterization systems. Nevertheless, it remains difficult for artists to design certain natural shapes, such as volumetric clouds due to their amorphous boundary structures.
In this section, a number of previous works of cloud modelling and rendering techniques are reviewed. Due to the cross uses of many specific techniques, we design a cloud modelling spectrum based on the level of intuitiveness to organize the previous cloud modelling techniques and our method, as shown in Table 2.1. In this spectrum, we describe three general methods that have been used to model and render clouds: procedural methods with limited shape control, procedural methods using proxy geometry and sketch-based methods. The procedural methods are mainly parameter-based cloud generation, which provide limited shape control for both artists and programmers. The artistic style represents the trend of intuitive interfaces that facilitate cloud modelling without much parameter understanding. Proxy geometry enables basic shape management by the placement of a large number of proxy geometries such as metaballs or boxes. Sketch-based methods allow cumulus clouds to be generated from a sketched silhouette, which is more user-friendly but type-limited. Our method fits into a new modelling style which aims to provide a better cloud modelling interface for artists.

2.1.1 Procedural Method with Limited Shape Control

Table 2.2: Representative procedural generation and rendering methods for clouds.

Citations	Modelling Method	Rendering Method
Reeves,1983	Particles	Probabilistic Shadow Determination
Nishita,1996	Metaballs	Splatting with light scattering
Dobashi,2000	Metaballs+Cellular Automation	Optimized Splatting
Perlin,1985	Noise Generation	Texture Synthesis
Elbert,1997	Implicit Noise Operations	Texture Synthesis
Gardner,1985	Particles+Ellipsoids	Texture Synthesis
Miyazaki,2001	Voxels+Atmospheric Fluid Dynamics	Optimized Splatting
Harris,2003	Voxels+Physics Simulation	Dynamically Generated Imposter
Dobashi,2010	Photo-based	Hybrid Rendering

Procedural methods are widely used in cloud modelling due to their realistic simulation of physically-based phenomena. In order to generate clouds efficiently, a variety of procedural methods have been introduced. Table 2.2 lists the primary representative cloud modelling techniques together with their rendering method.

Reeves [97] was the first to propose a particle system capable of producing objects with 'fuzzy' structures such as trees, fires and clouds. In this particle system, objects are considered as a group of primitives. The properties of these primitives, such as the position, size and color, are mostly managed in a procedural manner. For rendering, Reeves *et al.* [98] use approximation and probabilistic algorithms to visualize these particle objects. In their shading model, a single tree usually contains millions of independent particles. Instead of shading each particle precisely by calculating whether it is in shadow, a probabilistic shading model based on the particle's position and orientation is implemented to determine whether the particles should be highlighted or clipped. However, at that time, the drawing of these particles was so time-consuming that the performance optimization became a popular domain in the field of volumetric rendering.

One breakthrough is the 'splatting' method proposed by Westover [114]. In order to display the dataset in a regular volume, each voxel is drawn as a cloud of points (footprint) that spreads the voxel contribution across multiple pixels in the shape of sphere or ellipse. These footprints are then accumulated using forward mapping algorithms. This significantly improves the performance as these algorithms are easily parallelised. Although the original splatting method produces poor quality, a number of improvements have been developed to enhance the splatting quality, such as the *'Textured Splats'* by Crawfis *et al.* [20], *'Post-classified Splatting'* by Mueller *et al.* [81] and *'FastSplats'* by Huang *et al.* [53].

In cloud modelling systems, the splatting-based method is also widely adopted for cloud visualization. For example, in the cloud display system proposed by Nishita *et al.* [84], their splatting method takes account of global illumination. Each splatted cloud footprint is illuminated by the multiple light scattering.

For the cloud generation part, they propose a procedural method that uses metaballs (or blobs). A metaball is an isosurface defined by its center, radius and the density at

the center of the ball. Unlike mesh-based surface which requires a large number of vertices, a small number of overlapped metaballs can produce a complex curved surface, such as a cumulus cloud. Dobashi *et al.* [26] and Roditakis [99] extend this method of using metaballs to match clouds in real-world satellite images. They also extend the splatting rendering to support the viewing of 3D clouds from arbitrary directions. This is sufficient for scientific visualization purposes but is unable to accurately reproduce high-frequency effects, such as wispy striations along cloud boundaries.

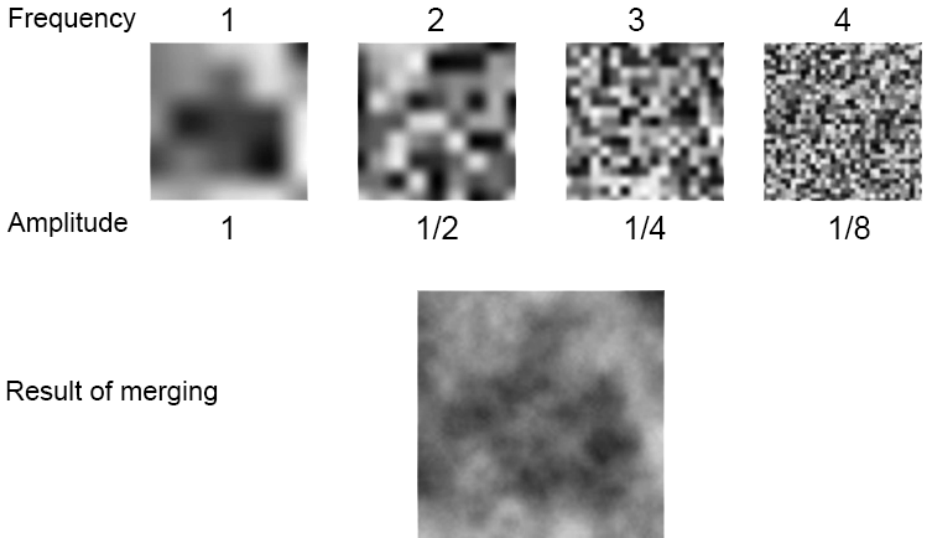


Figure 2.1: A cloud texture is generated through the merge of different octaves, which are the pseudo noise functions generated at varied frequencies and amplitudes.

Procedural noise, as proposed by Perlin [91] and Lewis [66], enables image synthesis with high-frequency details. It is widely used to synthesize the textures of complex surface patterns, such as wood, metal and clouds etc. Figure 2.1 shows a cloud texture generated using the noise method. The noise is created by an algorithm based on pseudo random number generation. Perlin produces a number of textures (Octaves) with the noise generation at various frequencies and amplitudes. These textures are merged to form the texture with a cloud pattern. Based on this, Ebert [31] implements a cloud modelling method using several implicit functions of noise operations. This enables the generation of different types of clouds by varying parameters. Our system implements a noise generation method to enhance the details on the cloud surface,

but one based on a simplex 3D noise algorithm [92].

To support the creation of various types of clouds, such as cumulus, stratus and cirrus, Gardner *et al.* [41] simulate the cloud on the surface of ellipsoids using a mathematical texturing function. This mathematical function is a simplified Fourier algorithm composed of short sums of sine waves. Clouds are simulated through the composition of multiple cloud ellipsoids, in which each ellipsoid contains an irregular transparent texture. Lewis *et al.* [66] combine this ellipsoid idea with noise generation functions to increase performance. With the use of multiple ellipsoids, Elinas *et al.* [34] modify Gardner's method to suit the OpenGL pipeline. This enables the rendering of various clouds at interactive rates.

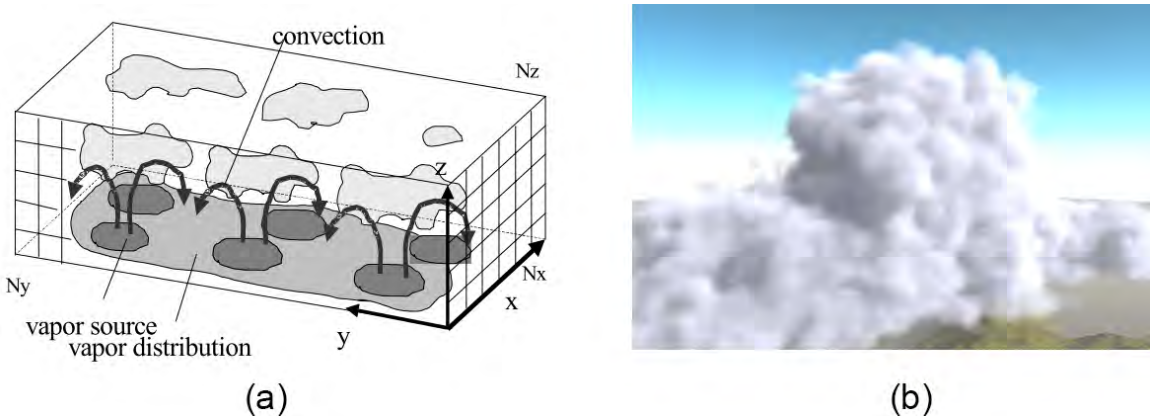


Figure 2.2: A voxel-based cloud simulation proposed by Miyasaki. (a) Using a Coupled Map Lattice to simulate Benard convection (b) The resulting simulated cloud.

However, these surface-based cloud generation methods have difficulty in simulating the evolution of clouds, as clouds in nature are not always static. Thus, another important approach is implemented based on a volume of voxels. Each voxel represents a single cell of a regularly divided 3D space. This normalized space structure significantly decreases the complexity of the representation of a volumetric object. Based on this voxel structure, Nagel *et al.* [82], Dobashi *et al.* [26] and Liao *et al.* [70] simulate clouds generation using cellular automata models. Figure 2.2 shows a cloud simulation system using voxel-based atmospheric fluid dynamics by Miyazaki *et al.* [79]. The fluid dynamics are implemented using a *Coupled Map Lattice(CML)* method proposed by Kaneko [59], which is an extension of cellular automaton with continuous state

values at the cells instead of discrete values. These systems are rendered based on an optimized splatting technique introduced by Dobashi *et al.* [26].

One major challenge of these voxel-based cellular automaton models is the extensive computational costs on cloud generation. Therefore, Overby *et al.* [90] and Harris [48] extend the CML simulations to a water vapor evolution model, which can be implemented on a programmable graphics hardware so as to increase the performance. Nevertheless, these physical simulations involve the control of the parameters for incompressibility, advection, vapor and temperature diffusion and buoyancy, etc. It is difficult for a modeller to understand and master these variables in a short time. Figure 2.3(a) shows a sequence of cloud generated on a 128×128 2D grid.

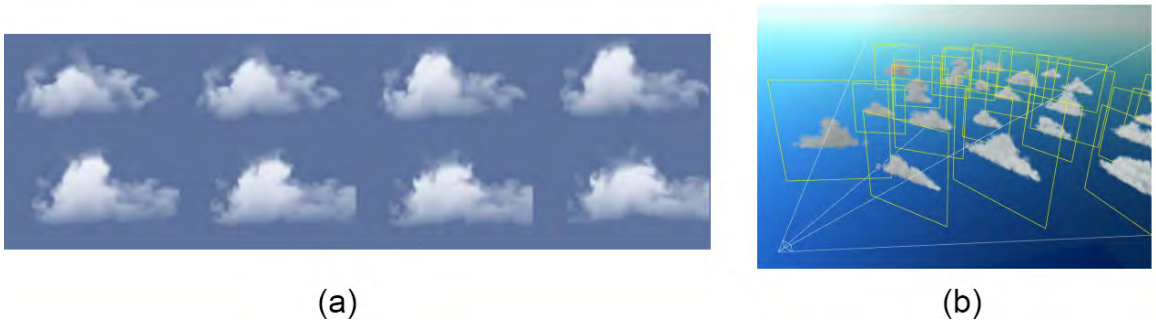


Figure 2.3: A physically-simulated cloud proposed by Harris [48]. (a) A series of cloud growth steps on a 2D grid (b) Using a Dynamically Generated Imposter technique for interactive rendering.

For interactive purposes, Harris *et al.* [47] introduce a real-time cloud rendering method based on the dynamically generated imposter technique proposed by Schaufler [102], as shown in Figure 2.3(b). This real-time cloud engine interpolates the densities between the cloud imposters based on the camera's position and view direction. Unfortunately, the generation of the imposters from physically simulated cloud data runs in off-line mode. Liu *et al.* [72] propose a scheme of dynamic cloud generation, which uses cellular automation to simulate the particle movements. In our system, the structure of cloud dataset is also a voxel volume in a 3D texture, which can be processed by the GPUs to achieve interactive performance.

Dobashi *et al.* [28] introduce another approach that simulates the clouds from a real photograph using a hybrid of various procedural methods. Figure 2.4 shows the results for the generation of altocumulus and cumulus clouds. In this method, cirrus clouds

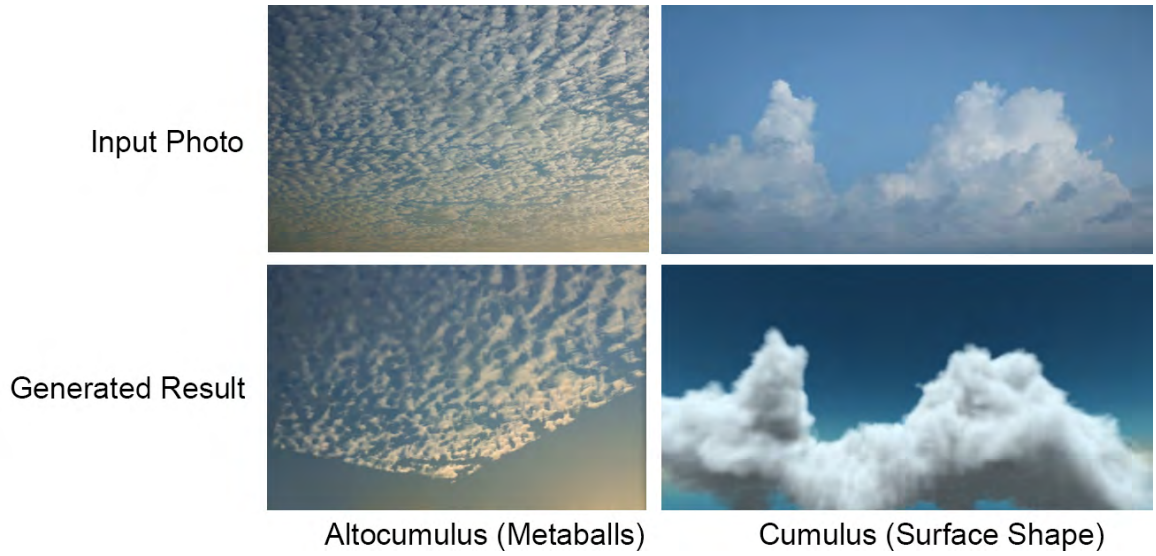


Figure 2.4: Photo-based clouds generated by Dobashi [28]. Note that different cloud types are generated using various procedural techniques.

are represented by 2D textures. Altocumulus clouds are simulated using metaballs and the cumulus clouds are generated through surface construction. In order to render the cloud with better color and light effects, Dobashi *et al.* [29] implement a parameter search method for color histogram matching.

Overall, these procedural methods provide a certain degree of shape control, such as the growth and dispersion of the clouds. Nevertheless, the formation of the cloud is procedurally generated, and this is difficult for artists to manage. In general, shape control is lacking. Our system, on the other hand, enables the artist to create a desired cloud in terms of shape, type and lighting.

2.1.2 Procedural Method using Proxy Geometries

In order to enhance shape control, and allow the modeller to create a desired cloud mass, Ebert *et al.* [32] outline a widely adopted strategy for cloud creation:

1. High level modelling of the rough overall shape
2. Low level automated procedural generation to grow detail around the proxy cloud.

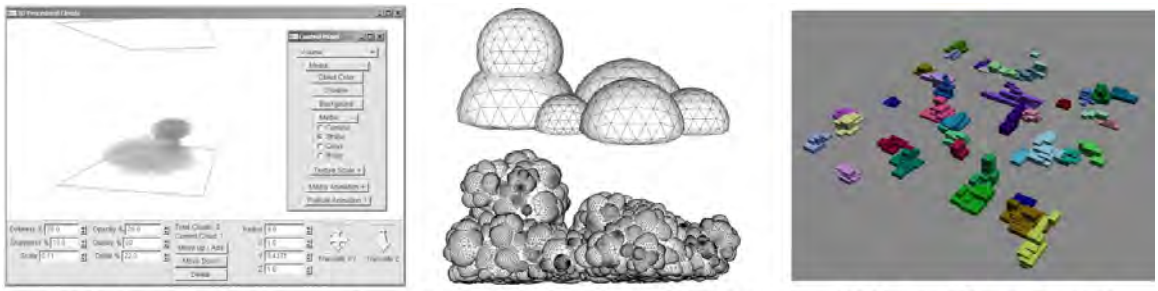


Figure 2.5: A series of cloud simulations with shape control using various proxy geometries.

Based on this two step strategy, Figure 2.5 demonstrates various geometries used to manage the shape of the cloud. Schpok *et al.* [103] propose a parameter controlled interface for cloud modelling and animation. In their system, several procedural methods, such as Perlin noise, are used for low level detail. At a high level, users are required to manually define the shape and location of proxy geometry, such as ellipsoids. They use an accelerated volumetric rendering method on texture mapping supported hardware, proposed by Cabral [12].

A related strategy is used by Bouthors and Neyret [10] to generate cumuliform clouds by procedurally blobs growth based on a user defined root shape. The rendering is based on Gardner's [41] ellipsoid texture method. In this case, cumulus clouds are able to contain high frequency detail with the coverage of mini blobs on its surface, as shown in Figure 2.5(b). Besides the limitation on cloud types, another restriction is the lack of user-management in the procedurally generated blobs. It is difficult for modellers to modify the shape once the cloud is grown.

To address this issue, Wang [111] uses a box-accumulation method to specify high-level shapes, which modellers are able to roughly design using existing modelling packages. At a low level, 16 distinctive textures are used to procedurally generate various types of clouds. Wang achieves a 100× speed-up over the similar rendering system proposed by Harris and Lastra [47], with some sacrifice of viewing angles. This cloud simulation system has been successfully adopted in many games, including Microsoft Flight Simulator 2004 [77].

Many existing modelling softwares also adopt a similar two step strategy for off-line

cloud generation. First, modellers are required to define the boundary of the cloud either by a shape control system, such as the mesh container in Blender [9], or a particle emission system as in Softimage [3]. Second, noise or particles are procedurally generated within the boundary to form the final shape of the cloud.
Unfortunately, this manual specification of proxy geometry remains time consuming. It is a particularly tedious task when modelling complex clouds with specific shapes.

2.1.3 Sketch-based

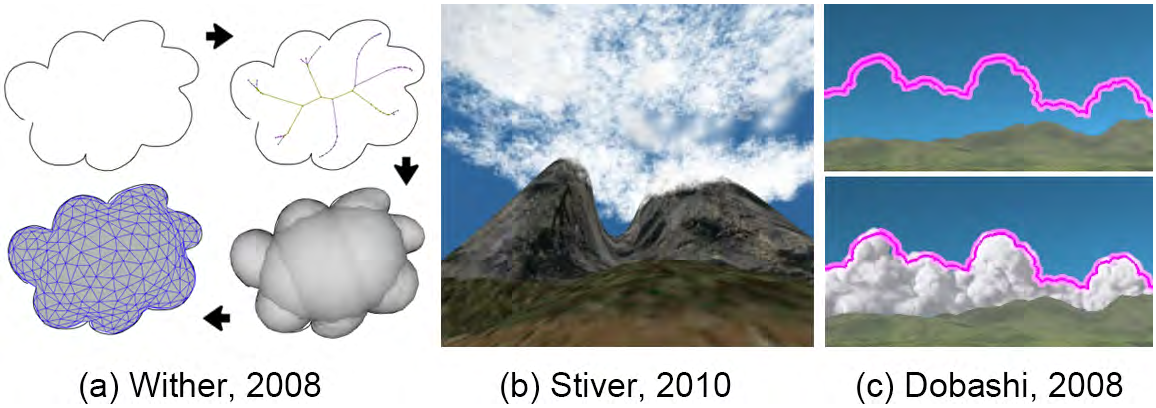


Figure 2.6: A series of cloud simulations using sketching methods.

Starting with the 3D freeform model design system proposed by Igarashi *et al.* [56], to create blobby objects such as stuffed toys by stroke drawing, Sketch-Based Interfaces for Modelling (SBIM) have become a trend. These methods allow the modeller to create 3D objects in a natural fashion, which has the potential to enhance productivity significantly.
In the field of cloud modelling, Figure 2.6(a) shows a sketch-based method for 3D cloud generation, proposed by Wither *et al.* [115]. They convert 2D silhouette strokes to 3D meshes based on an inferred skeleton (developed by Prasad [94] and Horst *et al.* [51]). They render the cloud meshes using a volumetric light scattering method [11]. According to their performance results, mesh generation can take up to 26 seconds. This method works well for small cumuliform clouds but becomes cumbersome for complex cloud shapes. Stiver *et al.* [106], (as shown in Figure 2.6(b)), extends this

method to support the modelling of other cloud types at interactive rates, which are rendered using a basic particle system without light scattering.
In Figure 2.6(c), Dobashi *et al.* [27] also adopt a sketching approach. In their system, users are required to depict the contours of the clouds from a specific camera position. Then, the clouds start growing using computational fluid dynamics (CFD) customized to allow shape control of the clouds. Their stroke mechanism limits the growth of the cloud to filling the envelope of the sketched shape. This method is limited to cumuliform clouds and generation takes several seconds for each growth step.

2.1.4 Common Problems in Cloud Modelling

The aforementioned techniques share a common problem: modellers often find it difficult to create volumetric clouds based on a desired shape. What is lacking is a consistent mechanism to model various types of clouds where the high level shape can be specified and the low level surface details can be preserved. In addition, the rendering of the modelled 3D clouds is either performed offline or introduces a lag of seconds or even minutes for cloud generation. This is fatal for an interactive modelling method. Therefore, we propose a new cloud modelling technique, a brush calligraphy style. This technique allows artists to model 3D clouds in an interactive and intuitive fashion.
Before discuss our system in further chapters, we give a brief introduction to the Chinese brush calligraphy, followed with related work on brush simulations.

2.2 Chinese Brush Simulation

"The writing mirrors the writer. The painting mirrors the spirit."
This Chinese proverb describes the use of the Chinese brush in a form of spiritual expression called the 'mind image' [38]. Using a Chinese brush is like carving using a knife. Calligraphers practise their brushwork in a very disciplined manner. Once they advance in managing brush control, they can project their minds intuitively into their writing. Unlike pen sketching, each stroke painted by the Chinese brush is

represented by various thicknesses and edge effects due to subtle changes in pressure and stroke flow. At this stage, we will highlight the intricacy and beauty of Chinese brush calligraphy through a series of historical examples. We expect this to help us better understand the features applied to our cloud modelling technique.

2.2.1 Brush Features with Historical Examples

(a) Emperor Liu (AD 58~88)

(b) Emperor Cao (AD 155~220)

(c) Emperor Zhao (AD 1082~1135)

Figure 2.7: Some calligraphy by the emperors across Chinese history. These emperors are Liu (AD 58–88), Cao (AD 155–220) and Zhao (AD 1082–1135).

Dynamic Thickness Along the Curve

Figure 2.7(a) shows one of the earliest discovered handwritings by Emperor Liu (AD 58–88), which means 'the stars are spreading over the sky'. The main feature of this calligraphy is captured in its third character, which contains a large curving stroke. The font that the Emperor Liu uses here is known as 'Running' style or 'Curving' style. The use of curves enables a link between two consecutive strokes, swiftly and naturally. Additionally, a single curve can also been used as a short form to represent a combination of numerous strokes. For this third character, the regular writing style is 列, whose middle and right parts are reduced into one curving stroke in this case. We notice that using curves not only speeds up the writing but also enhances the creative freedom for the artist as they are no longer limited to straight lines. Moreover,

all the strokes depicted by Emperor Liu, both straight lines and curves hold different thickness along the stroke. This variety in thickness is the first feature in using the Chinese brush: it is not only dependent on pressure, but also affected by the speed of the movement and the flow of the brush.

Importance of Stroke Flow

The stroke flow can be considered as the path of the brush movement. Figure 2.7(b) illustrates two characters in a new font style named 'Li', a clerical writing style. This calligraphy was written by Emperor Cao in AD 219 [44]. It means 'Rolling Snow'. Li [68] describes how the left character 'Snow' presents a 'Static' impression through its solid strokes. The right character, which means 'Rolling', provides a more 'Active' emotion. This sense of activity is delivered through its Right-Falling stroke, which possesses the figure of a rolling wave.

Different emotions can be conveyed by different stroke shapes. In this case, the general purpose for using a bold font such as 'Li' is to present an artistic impression of steadiness. Additionally, different stroke flows produced by an individual deliver various artistic feelings. It is similar to the different feelings evoked in an audience when they experience the same piece of a symphony performed by a different orchestra.

Power from the Bending and Hook

Figure 2.7(c) presents a poem 'Summertime', which describes a view of a garden on a summer day. It is versified by Emperor Zhao who created this new font style in Chinese history. This font has three features, namely: slim strokes, few curves and sharp bending and hooks at the end of strokes. The sharp bending and hooks are produced by the pressure change when the stroke flow alters. This pressure changing involves a precise coordination between fingers, hands, wrist and arms.

This precise control of power, conveyed from the arm to the tip of the bristle, transforms into the final brush footprint in an elegant way. Thus, according to Dan's study [21], Emperor Zhao initially named this new font as 'slim tendon', since the tendon is the source for strength in ancient Chinese philosophy. Finally, in order to show respect

for this elegant font style, he converted the name from ‘tendon’ to ‘gold’ since they are homophonic characters.

Fractals from Movement Speed

Figure 2.8: A character ‘Loong’ (a Chinese dragon) written by different emperors. (a) Emperor Kang (AD 1654–1722) (b) Emperor Guang (AD 1871–1908).

Figure 2.8 illustrates two examples of calligraphy written by two emperors: Emperor Kang and Emperor Guang. These two calligraphies are of the same Chinese character ‘Loong’, which is a legendary Chinese dragon symbolizing divine power. Both these two calligraphy use a bold font style. Strokes in Figure 2.8(a) are straight and concise. These straight lines form a beautiful balanced structure, which presents a standing Chinese dragon showing its power.

On the other hand, Emperor Guang’s writing as shown in Figure 2.8(b) reflects another side of loong’s character, the vigorous vitality. Its curved strokes generate a fractal effect, which creates the association that the loong rushes into the air to tear the sky open.

This fractal effect, which is the broken stroke in the middle of the character, is produced by the fast movement of the brush. The artist draws the stroke swiftly so that only part of the ink disperses into the paper. This unique feature can produce a vivid wispy effect that enhances the high-level details of the clouds surface, in particular for

cirrus clouds.

Clouds in Landscape Painting

Figure 2.9: Various clouds painting methods in Chinese landscape paintings. (a) Album of painting by Yun Ming Shan Ren. (b) Z. Kong, 2000.

Chinese landscape painting is always inscribed with a short poem and the author's name in a calligraphic style. Hu [52] discusses that these two art forms have the same structure as they use the same tools and similar skills. Rawson [96] points out that one common feature in landscape paintings is the ubiquitous clouds, which enrich the artistic conception through ambiguity. Lin *et al.* [71] generalize several prevailing techniques in painting clouds using the Chinese brush. One old technique is a reversal painting method (Negative Space), in which the surrounding objects are first painted such as mountains and rivers, and this leaves the empty space for cloud painting. Once the surrounding objects have dried, a wet brush with clean water is dipped along the edge of the mountains and rivers to form a diffuse cloud effect, which is shown in Figure 2.9(a). Figure 2.9(b) shows another important method for cloud painting. It uses dark strokes to illustrate the silhouettes of cloud. Strokes are bold, especially at the bottom of a cloud. This provides a dark shadow effect which enhances the visual quality.

Mastering these cloud painting skills requires years of practise. By using digital devices, our cloud modelling system adopts a simple straightforward mechanism, which is

difficult to implement in conventional Chinese landscape painting. We will further explain the new mechanism in the Chapter on Brush Simulation.

Feature Summary

Calligraphy is similar to wine tasting, which is not only about savoring flavor, but also a taste of condensed time. Calligraphy, a writing style formed through years of disciplined training, also presents a taste of history connected with people. In fact, we found it very difficult to separate the spiritual beauty from the calligraphy. However, we are strongly motivated to study the features from their footprints. These features we summarise as:

1. Diversity: various thickness and flows shapes.
2. Simplicity: simple stroke and simple color.
3. Intricacy: sophisticated bending and fractal effects.
4. Beauty: a balanced structure to the characters.

In the digital era, there have been numerous studies around the simulation of this ancient art of painting using modern digital devices. The next section provides an overview of this work.

2.2.2 Brush Simulation on digital devices

The simulation of landscape painting normally involves two aspects: brush deformation and ink dispersion simulation. The first attempt in simulating the brush was proposed by Strassmann [109] in 1986. He used a splatting method for footprint rendering so as to produce various thicknesses in one stroke. In his environment, users were required to manually specify a number of control points and pressure levels to generate each stroke. This leads to an unnatural painting style that severely impacts the user experience. In addition, the rendering of each stroke takes one to two minutes on average making their system difficult to use. However, the basic four system components, the simulation of brush, stroke, paper and ink, are widely adopted. Thus, to simulate brush calligraphy in high fidelity, various methods have been developed around the simulations of these

four components.

Lee [65] proposes a fiber mesh structured paper and ink flow. Together with a physically-simulated 3D brush [64], users are able to generate black ink with a diffusion effect at interactive rates. This provides a more natural interface but is still limited in capturing the wispy effects at the edge of the strokes. Since then, there has been extensive study of brush calligraphy using physically simulated methods.

Physically Simulated Methods

Baxter *et al.* [5] develop an interactive haptic system for general brush simulation. They employ a spring-based skeleton attached to a mesh surface to mimic the deformation of the brush. In order to simulate the unique features of Chinese brushes, such as bristle forking, Yeh *et al.* [120] improve Baxter's model by using a spring-based skeleton with 9 bristles to simulate the effects of pressure and friction. Both of them use a PHANTOM force feedback device to control the brush with haptic perception. For ink simulation, they implement a multi-layered structure that models the storage and transfer of ink between the brush and rice paper. Xu *et al.* [118], Baxter *et al.* [7] and Chu *et al.* [14] also propose a 3D brush with a skeleton-based structure, Figure 2.10(a). However, they use a more affordable device, in which Chu binds gyroscopes and ultrasound distance sensors to a real Chinese brush, for input. In their ink diffusion model [15], a lattice Boltzmann equation simulates ink dispersion in absorbent paper. To increase the realism of the ink effect, Huang [54] proposes a more complex algorithm based on an observational model, which produces an impressive diffusion effect. However, a fully dispersed stroke requires minutes of ink diffusion calculation, which makes it impractical for interactive purposes.

With the development of programmable graphics cards, many systems implement the physical simulation on GPUs to speed up performance. Baxter *et al.* [6] and Chu *et al.* [16] optimize performance of their 3D brush simulation and create an oil painting engine, which is used in Microsoft's Fresh Paint application [78], Figure 2.10(b). Lu *et al.* [74] propose a GPU-based method that simulates thousands of individual bristles in the form of Bezier curves as shown in Figure 2.10(c). We also implemented a similar method in our skeleton-based 3D brush that leverages the parallel power of graphics

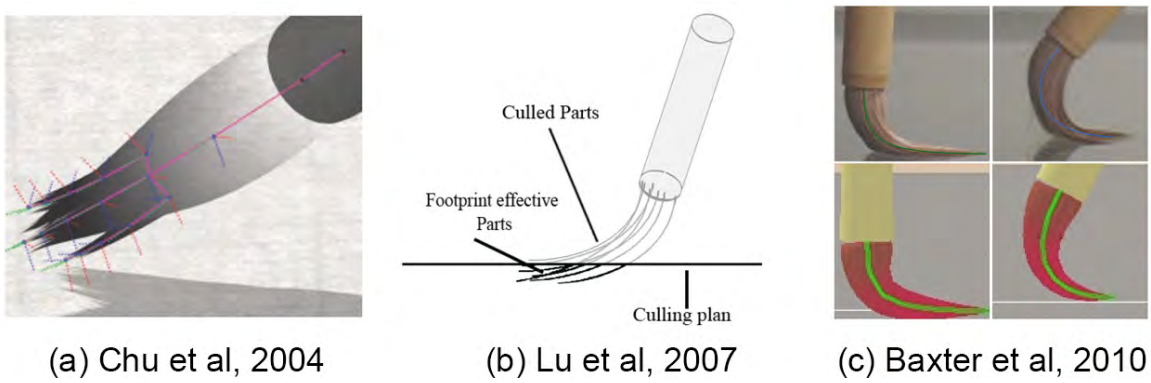


Figure 2.10: Various physical 3D brush simulations.

cards.

Procedural Methods

Instead of employing complex 3D brush simulations, procedural methods can also produce persuasive results at a lower computational cost. DiVerd *et al.* [25] [24] present an interactive paint algorithm that procedurally mimics the ink diffusion effect through a particle system. The particles of the pigment are represented in the form of vectors, which allows rendering at arbitrary resolutions.

This paint engine is used by commercial packages such as Adobe Photoshop [1]. It is lightweight enough to be implemented on low-powered devices such as an iPad. The simulation result is currently limited to the water-ink effect. It is able to disperse the ink smoothly but is unable to generate the fractal pattern, which is important for representing various clouds with complex edge effects.

Texture Mapping Methods

Simhon *et al.* [105] and Okabe *et al.* [85] suggest a method that emulates footprints using Hidden Markov Models (HMMs). This module refines the input stroke lines with the mapping of a set of textures. With an automatic estimation of the footprint flows, brush deformation and drawing styles, it produces aesthetic strokes in 'Straight Stroke' style, as shown in Figure 2.11(a), at an average 8.7 seconds on a Linux system

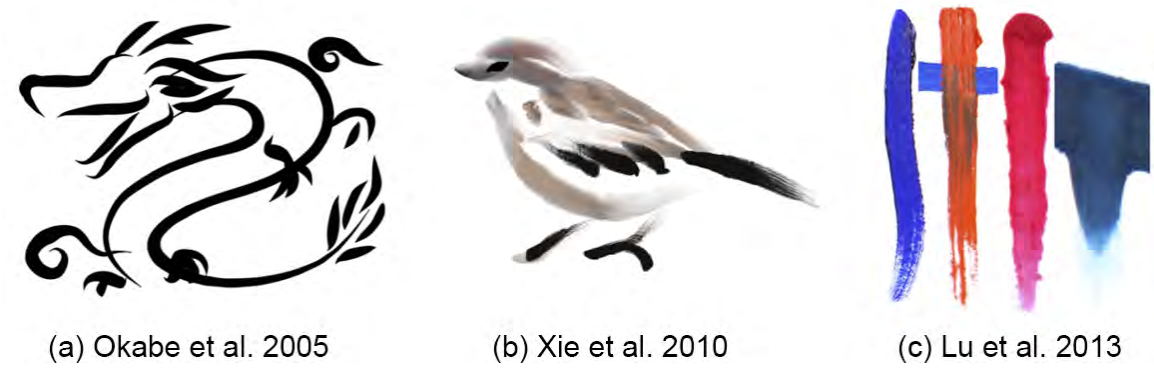


Figure 2.11: Various texture mapping methods in brush paint simulation.

with the Opteron 244 processor.

Similar to this, Xie *et al.* [117] use contours to represent the stroke shapes. The system automatically estimates the brush trajectory from the outline of the stroke shape, which is able to produce an oriental ink painting as presented in Figure 2.11(b). Recently, Xu *et al.* [119] extend this technique to render geometric models using an ink-wash painting style that presents an oriental visualisation of 3D objects in real-time.

Kim *et al.* [60] and Lu *et al.* [73] developed paint systems that use scanned images of real natural media to synthesize the footprint without the implementation of any procedural methods or physical simulations. Figure 2.11(c) presents the result from the Lu's RealBrush engine. It adapts to various painting styles from oil painting to water-ink painting using customized textures and an embedded smearing and smudging tool.

Unfortunately, such high fidelity simulations are problematic from our perspective. Simulating accurate bristle spread and ink dispersion is computationally expensive, particularly when combined with volumetric rendering. Instead of a precise physical simulation with specialized hardware, our aim is to reproduce, using a stylus pen for input, the main characteristics of a Chinese brush that make it so effective for cloud painting.

2.3 Summary

In Chapter 1, we present three major types of clouds in nature, which are cumulus, stratus and cirrus. The generation and visualization of these clouds in digital world is largely different from the traditional method.

This chapter first reviewed the various techniques in the digital field of cloud modelling and rendering. Since many of the cloud modelling methods involve a mixture of techniques, such as a hybrid of particles and metaballs, we organize these methods with regard to their user interface from a completely procedurally controlled method to a sketch-based style. These modelling and rendering methods are able to generate realistic 3D clouds but there remain problems such as inefficient shape control for various cloud types and low performance for interaction purposes.

The second part of this chapter is an overview of brush simulation techniques, which is led by a demonstration of several ancient Chinese calligraphic works and landscape paintings. These digital brush painting methods are able to simulate brush features such as the various footprint patterns and ink dispersion effects. These effects are able to demonstrate the beauty and diversity of calligraphy, which can also be leveraged for cloud modelling. However, we do not expect to simulate the brush work with high fidelity due to the computational cost.

In order to provide an efficient and intuitive 3D cloud modelling method from the artist's perspective, the structure of our new system is designed to follow the workflow of traditional painting styles. Artists are expected to paint the cloud in conventional painting style and our system then produce the desired volumetric clouds in an instant. In the next chapter, we will discuss the framework of our system.

Chapter 3

System Structure

This Chapter outlines the structure of our cloud modelling system. Since the final structure is very different from the initial design, we will discuss the workflow in detail to elaborate on our design decisions.

3.1 Modelling from the Perspective of Artists

In section 2.2.2, we presented several state-of-the-art modelling systems that generate clouds in various ways. The procedural methods [48, 26, 91, 97] require the tuning of a number of variables. It is difficult for modellers to understand the meaning of these parameters and their working mechanisms. In addition, off-line generation makes the tuning of parameters time-consuming. In many cases, modellers are annoyed by having to wait for several minutes only to find that the result of the generated clouds are significantly different from the desired shape. Furthermore, due to the limited control of the procedural process, most parameter adjustments require the process to be restarted from the beginning. This repetition of parameter setting, procedural generation, waiting and shape checking, is a tedious process that impedes the creativity of aesthetic cloud modelling. Despite the rapid development of hardware, and procedural methods consequently becoming faster and more interactive, the lack of shape control remains a weakness.
The extended procedural methods with proxy geometries [32, 103, 111] improve the

interface through the support of the shape control. In spite of support for building desired cloud shapes, these methods not only require parameter tuning but also involve a large amount of work in shape building by placing a number of proxy geometries in a 3D world.

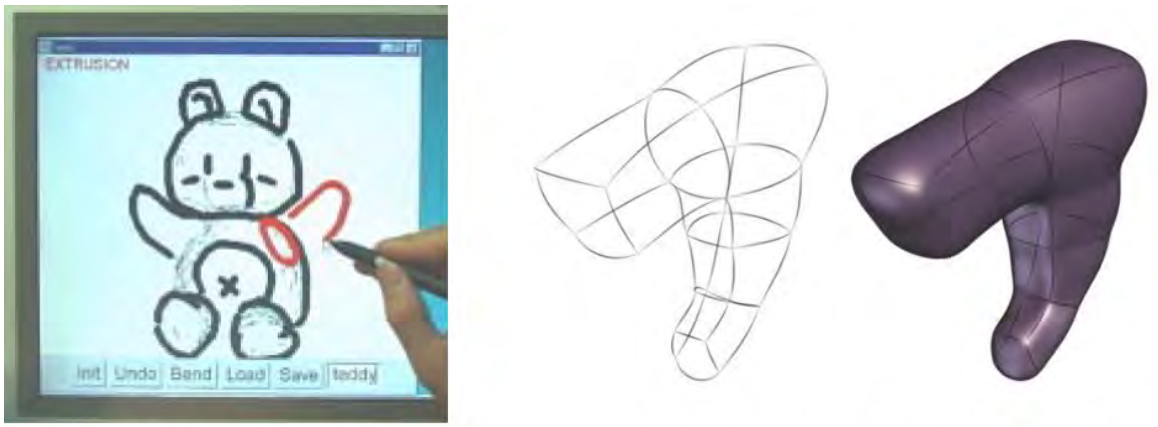
Recent, sketch-based systems [115, 106, 27] provide a more intuitive interface. The large number of obscure variables are concealed from modellers. The process of cloud modelling can then concentrate more on shape design. Unfortunately, the number of generated cloud types using such interfaces is limited. Furthermore, many clouds in nature have no clear boundary and cannot therefore be sketched precisely.

Nevertheless, these methods, from procedural generation to sketch-based methods, lead the trend towards a more user-friendly interface, simulating the way in which an artist works, which is more intuitive to use and quicker to learn.

We consider modelling from an artist's perspective to be a promising approach and extends this approach to support more cloud types with a better user interface. Our work has three major questions that must be considered when converting from a 2D canvas to a 3D model:

1. How does one grow the shape drawn on a 2D canvas into a convincing 3D object?
2. How does one locate the 3D position of an object that only has two dimensional properties?
3. How precise should the extent of growth and location be?

Figure 3.1 shows two solutions which have been developed to tackle the ambiguity of extending from 2D to 3D. The Teddy system proposed by Igarashi *et al.* [56] provides a planar canvas that grows a closed curve to a rotund object. Das *et al.* [22] and Orbay *et al.* [89] use a 3D network of curves to reconstruct the edges of an object along the depth coordinate. This provides a certain level of depth generation, which makes it possible for 3D modelling from a 2D sketched object. Instead of interpreting the 2D sketches as 3D curves, Dorsey *et al.* [30] present an approach that organises a set of 2D strokes on a number of manually-specified planar canvases. It provides a fast way of building conceptional 3D objects from 2D strokes with low fidelity. However,



(a) Teddy by Igarashi et al. (b) Curve Networks by Orbay et al.

Figure 3.1: Various sketching methods. (a) A planar canvas is used to expand a closed curve to a rotund object. (b) A 3D curve network is applied to generate an 3D object along the depth coordinate.

the fundamental problem of space location determination from a sketched 2D object remains challenging, because, as Hoffman [49] comments, "the image at the eye has countless possible interpretations".

In the survey of sketch-based modelling systems produced by Olsen *et al.* [86], this major challenge is addressed through the notion of perception. In reality, viewers have their own visual rules that help them to perceive an imaginary 3D world from an arbitrary painting. These visual rules, which are practised through recognition and reconstruction, enable the viewer to understand objects given the rough information of their 3D shapes and locations.

A correct space location determination is crucial in our cloud modelling system, as clouds are commonplace in the sky. Therefore, we study the visual rules from the perspective of artists so as to help the design of our system structure.

Most landscape paintings or photos are created through perspective views either from the ground or from a high altitude. In particular, cloudscapes are seldom illustrated from an orthogonal view such as a top view or a side view. A technique of aerial perspective is applied in many landscape painting so as to enhance the viewer's perception of distant objects [8, 55]. When appreciating a painting or a photo with cloud features, viewers first identify a region of skyline and then recognize the coarse

3D location of the cloud by its type (cumulus in low altitude while cirrus in high altitude) and its relative distance to the other painted objects. Afterwards, they can naturally reconstruct the cloud in an imaginary 3D world based on its shape, size, type and shading. Although the result of the imagined reconstruction are hardly the same as the real cloud, viewers are generally less concerned about the accuracy of the reconstruction but rather about the shape of the cloud from their viewing direction and the colour it presents. This visual rule implies that clouds can be approximately perceived in a perspective view.
Therefore, a sketch-based prototype is built to assist in the evaluation of the following hypotheses:

1. Clouds are amorphous objects that are constantly changing. Their nature is such that it is difficult and unnecessary to shape and locate them as precisely as general objects with solid surfaces.
2. Viewers can approximately perceive the location and the shape of the clouds from an arbitrary painting of a skyline.
3. We may coarsely interpret the 2D painted clouds into a 3D world from a perspective view of the artist.

3.2 2D to 3D Mapping Prototype

In order to assess the feasibility of interpreting clouds from an artist's perspective, we developed a low-fidelity sketch-based prototype. It is worth noting that the main purpose of this prototype is to evaluate the feasibility of the 2D to 3D mapping. The implementation of the sketching style through the mouse device is intended to facilitate cloud data creation in a 2D painting view.
This prototype is implemented in C++ with the API support of DirectX 9.0c. It supports the following functionality:

1. In Figure 3.2(a), a stroke is generated along the mouse movement while the left mouse button is pressed. This simulates a coarse sketching style by means of a simple mechanism.

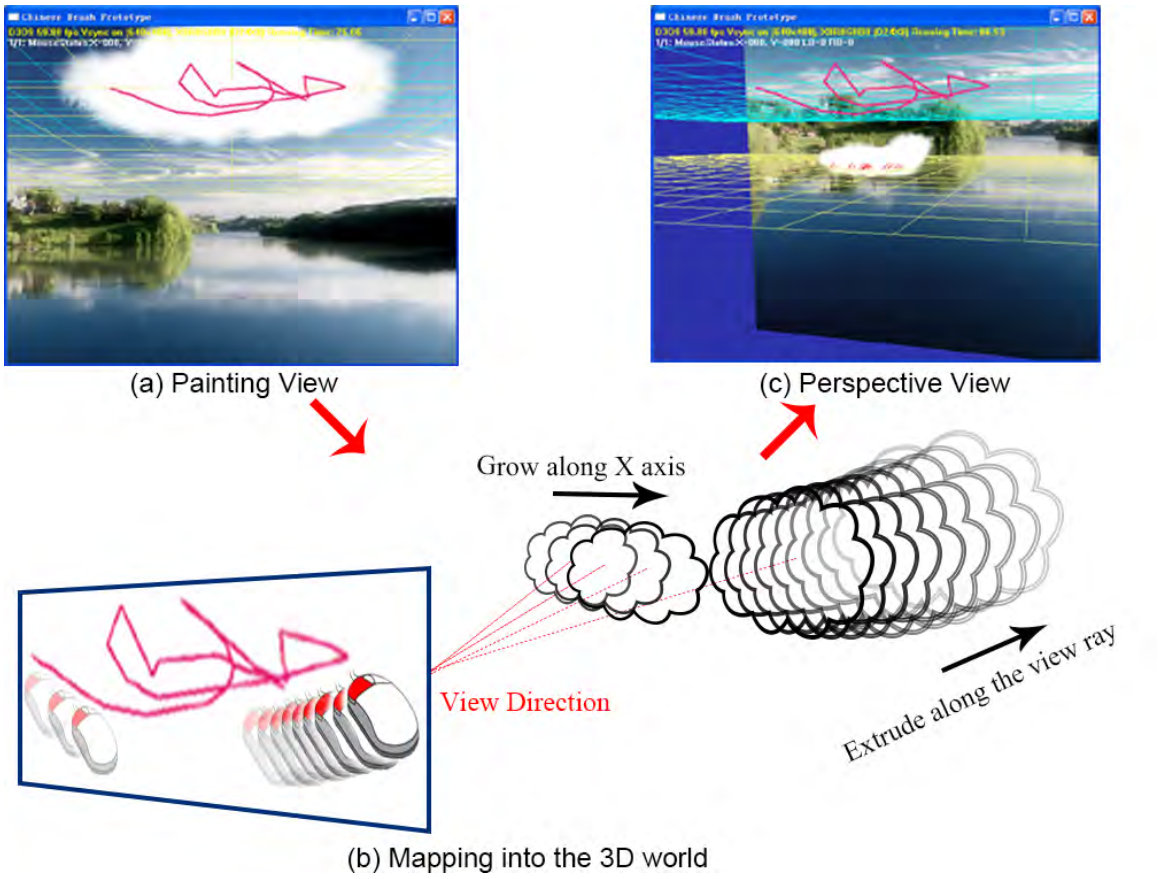


Figure 3.2: A sketch-based prototype that uses a set of 2D textures to present 3D clouds in the sky. (a) A stroke is drawn from the painting view. (b) A fast mouse movement leads the sample textures growing along the x axis of the viewing plane. Once the mouse halts for more than 0.1 seconds, cloud textures are extruded along the view ray. (c) The generated 3D cloud from a perspective view. Note that the blue and yellow grids represent the skyline.

2. Figure 3.2(b) shows that the stroke produces a number of sampling points with uniform distribution. A set of varied cloud textures are blended on these sampling points.
3. In order to produce a volumetric cloud from the 2D painting view, as shown in Figure 3.2(c), we attempt to extrude extra sampling points along the view ray, which is emitted from the current eye position and directed through the mouse picked pixels in the view plane. The number of extruded points is related to the duration during which the mouse halts while the left mouse button is pressed. It simulates the depth control of the generated 3D cloud via the pressure of a pen.

Figure 3.2(c) shows the perspective view of the 3D cloud, which is generated from the painting view in Figure 3.2(a). Note that the right part of the painted cloud grows along the view ray, which is upward-directed along the z coordinate.

Six postgraduate participants with an HCI background were invited to test this prototype. They were required to draw several strokes from a fixed view angle, as shown in Figure 3.2(a). After checking the generated 3D clouds from another view, the perspective view, they were asked to compare the result with their imaginary 3D clouds. Many of the students expressed their concerns about the regularity formed by depth extrusion in a straight line (view ray), as this leads to an unrealistic visual effect. With a clear identification of the skyline, the positions of the sampling points and their growing directions in the 3D world generally match expectations, but the method for cloud growth needs to be further refined.

In addition, this prototype exposes many problems in relation to the cloud structures. Our initial imposter-based cloud structure has similar drawbacks to the ones in Wang's [111] system:

1. The billboard technique, which is used to render overlapped 2D cloud textures, produces severe artifacts, particularly when the viewing angle is vertical to the skyline.
2. It is difficult to process the shading of the 3D cloud due to the lack of interior connections between the 2D textures. It is also impossible to incorporate important visual features, such as volumetric light scattering and 'rim lighting' using texture-based methods.
3. Cloud shape control is limited. Growing the cloud to a desired shape can be achieved by overlapping the 2D textures. However, it is difficult to implement an erase function during cloud editing. Fading the 2D textures is one alternative way of producing a dispersion effect. However, erosion at specific locations requires precise manual modifications of individual 2D textures, which is challenging when working with a large number of textures.
4. It is difficult to accurately manage the edge features, such as a hairy or mushroom style. Although various cloud samples are implemented, the sketch mechanism has limited control in the determination of these features at desired locations.

Nonetheless, the specification of edge features is important for the modelling of characteristic clouds such as an animal or any other specific shapes.

Therefore, we adopt a voxel-based data structure in later development. The voxel-based data structure has been used in many smoke and cloud applications [79, 48, 123, 18]. It not only matches the cloud structure in nature, but also provides support for dynamic shape changes with the help of procedural methods. Although it is more complex to render this voxel-based structure than a particle or polygon-mesh system, an increasing number of techniques have been developed to improve its rendering quality and the performance.

3.3 Voxel-based Data Structure

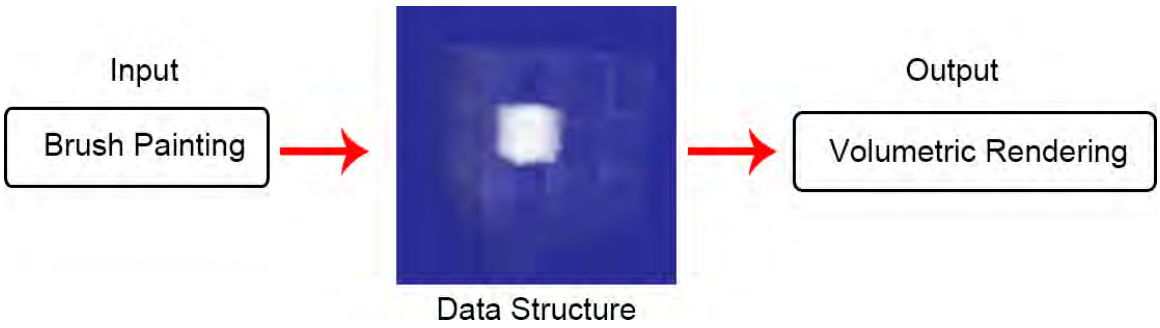


Figure 3.3: The structure of our system. The input method produces a voxel dataset which is rendered directly through GPU-based volumetric rendering.

Figure 3.3 illustrates the structure of our system. The core component is a voxel-based data structure, which is the middleware that connects the input and output modules directly. The dataset of the volumetric cloud is a 3D texture of 256^3 size, with the 32 bit float value of each texel representing cloud density at that location. Thus, each 3D texture occupies 64MB of memory, which is reasonable for both GPU and CPU operations. During execution of the application, varying numbers of 3D textures are employed for noise generation, octree processes and cloud compression, etc. This results in dynamic memory consumption of between 300MB and 500MB. Major performance bottlenecks occur during the transfer of data between CPU and

GPU. In order to reduce the impact on the user experience caused by such subtle performance drops, a number of stratagems are implemented such as the rendering of the 3D scene into a 2D texture so as to preserve computational power for brush simulation. A common strategy is to distribute the tasks into balanced workloads so that the computation of both the CPU and GPU can be leveraged.
In Figure 1.6, the voxel data structure is connected to two components which are also main modules of our system. A Wacom Intuos4 A4 tablet is used as the input device, although any stylus pen or tablet capable of registering tilt and pressure would suffice. Using this device, we simulate the brush calligraphy style and convert the footprint into a correlated 3D dataset. This 2D to 3D conversion requires CPU assistance which leads to a performance drop, but the generated 3D dataset is processed directly in GPU memory. With the use of GPU-based volumetric rendering techniques, the 3D texture in the GPU's memory can be immediately visualized. In this way, compelling 3D clouds can be modelled and presented at interactive rates.

3.4 Workflow

In general, the way in which a system is perceived by the programmer is substantially different from how the artist perceives it. Most artists are concerned more with how to use the system instead of how the system works and rightly so.
Figure 3.4 illustrates the workflow of the user scenario, which is implemented in our final system. As with preparing a canvas in reality, the user is required (as shown in Figure 3.4(a)) to decide on the view direction and the altitude of the 3D canvas (rendered in a darker blue). Subsequently (Figure 3.4(b)), the user paints a stroke of varying thickness onto the 3D canvas using our brush simulation method, which is discussed in Chapter 5. Once the stroke is complete and the stylus pen is lifted from the tablet, a hybrid mapping (Figure 3.4(c)) is undertaken to populate the stroke footprint with circles that are then mapped as spheres onto the canvas. The results are modulated by 3D noise and additional voxels are packed around the edges to more faithfully match the painted stroke. All in all, this process takes less than a second on average to generate a realistic volumetric cloud with high-frequency details on its

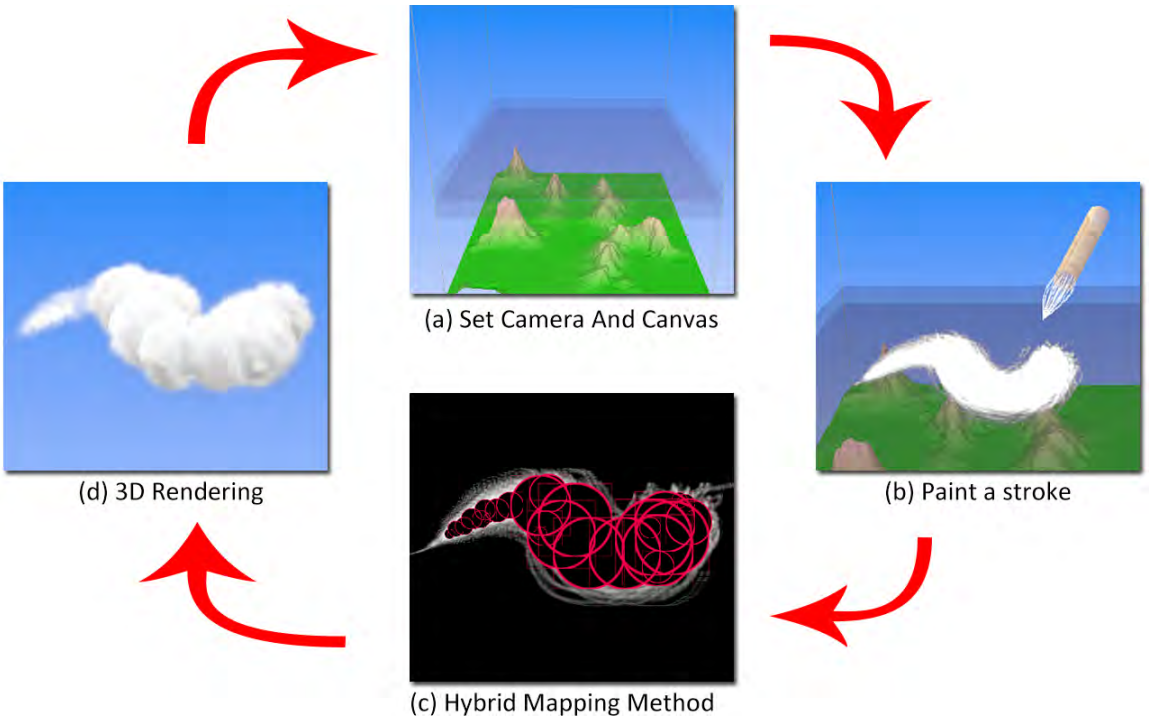


Figure 3.4: The basic workflow of our final modelling system: (a) The user specifies the camera view and canvas height; (b) A stroke is painted; (c) A cloud is generated using circle growth in our Hybrid Mapping Method, which takes less than a second on average per stroke; (d) The final 3D cloud with volumetric lighting is rendered.

surface. This cloud mapping and growth method are explained in Chapter 6.

Figure 3.4(d) shows the final generated 3D cloud, which is displayed using GPU-based real-time volumetric rendering. Volumetric lighting is also implemented to approximate the interplay of light and shadow so typical of real clouds. Chapter 4 describes our volumetric rendering technique, which incorporates a GPU-based octree process for empty space skipping and a volumetric illumination method. As a further affordance the user is able to adjust the light source location and colour interactively. The process of adjusting canvas height and camera view direction, as well as painting and volumetrically mapping strokes continues until the user is satisfied with the final result.

One of our key design decisions is the use of the 3D canvas as a projection surface. This establishes a certain altitude range for any given brush stroke. In reality, of course, clouds tend to fall within altitude bands and so this restriction has a natural

analogue. In any event, the 3D canvas can be raised or lowered between strokes.
However, a good modelling system requires more than the creation of 3D objects. Although this study focuses on the usability of cloud generation using a brush calligraphy style, a poor UI may affect the users' experience during the evaluation. For instance, error correction is a well-established tenet of effective interface design. Unlike 2D painting, correcting the current 3D cloud model is determined by both the camera direction and 3D canvas altitude. This involves the operations of multiple elements such as the rotation and translation of both the 3D camera and canvas. In support of this we implement some essential functions to improve user interactions: Erase Mode, Light Mode, a Menu System and a History Track function.

3.5 Erase Mode and Miscellaneous Functions

Erase Mode is a mode frequently used by the users. It allows users to erase a certain part of the cloud data and immediately regenerate the entire 3D cloud with volumetric illumination. This function is useful in any modelling system, as it enables data modification without rebuilding from the scratch.
In Chapter 2, we explained that with imposter-based cloud rendering systems [48, 111] it is difficult to perform data modification and rendering in real-time. They require a middle stage to convert the volumetric dataset (either voxel-based or particles) to a series of view-aligned imposters.
Unfortunately, this conversion stage is generally computationally expensive and converts from a 3D dataset to 2D imposters only and not vice versa. In our system, the 3D cloud is a voxel-based dataset stored in the GPU's texture memory, which is visualized through our volumetric rendering technique without any conversion. Thus, each texel of this 3D texture can be directly manipulated using the Compute Shader. Modellers are able to erase the cloud and render with real-time performance.
We implement two erase styles to modify the cloud data: a *Vacuum* style and a *Rub-out* style. They provide different cutting shapes for cloud modification.

3.5.1 Vacuum

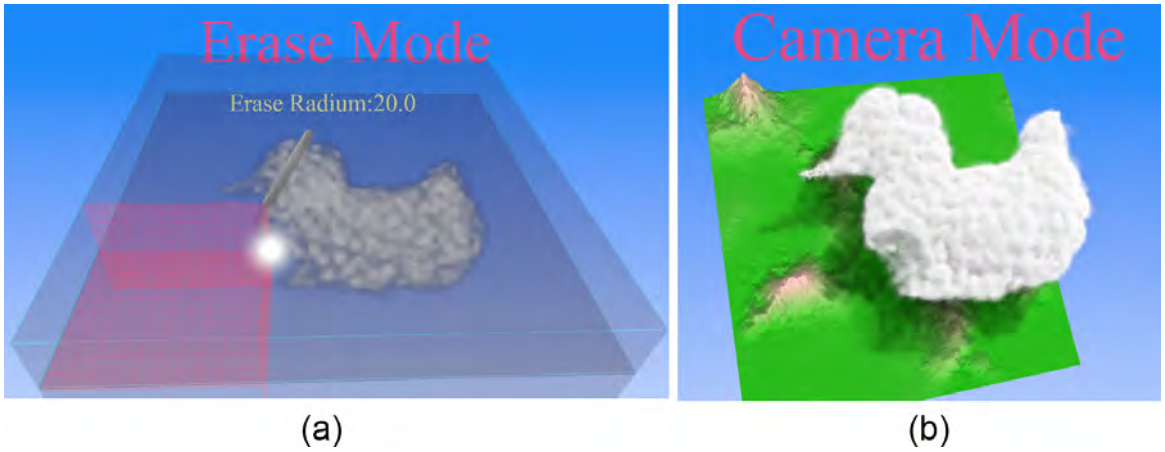


Figure 3.5: The Vacuum style in the Erase mode. (a) The location of vacuum point is defined by the pen's movement (XZ plane) and pressure (Y axis). The diameter of the sphere is controlled by the '-/+' key and the erase is triggered by 'space' key. (b) The Vacuum style produces a spherical hole with a smooth boundary.

The Vacuum style is a sphere-based cutting approach. Modellers are required to provide 3 coordinate position values and the diameter of the cutting sphere. Figure 3.5 demonstrates two elements we implemented to enhance the location representation of the cutting sphere. One is a metal rod and the other is intersecting grids from three axis-aligned planes. The x and y location values are related to the pen's movement on the tablet, while the height of the cutting sphere is defined by the pen's pressure. The 'Space' key triggers spherical cutting from the 3D dataset.

This erasing style aims to provide accurate modification so that modellers can achieve a desired shape. Unfortunately, it was not as well received as expected in a pilot test of 4 users. Users found it difficult to control the exact sphere location in the 3D canvas and its diameter with the use of both the pen and keyboard. Secondly, the hole generated from the Vacuum style contains a smooth spherical boundary, which also leads to an unrealistic cloud appearance. Thus, an alternative erase style was developed to help the shape management, Rub-out style.

Figure 3.6: The artist uses the Rub-out style to depict the eye of the duck cloud. It instantly regenerates the cloud data with 3D illumination using a Compute Shader.

3.5.2 Rub-out

The Rub-out style is a ray shooting approach. A modeller only needs to move the pen to the desired cutting location and press the tip; the cloud data inside the 3D canvas is then removed inside a cylindrical region which follows the viewing direction. The diameter of the cylinder varies with dynamic pressure control. Users use this erase approach more often than the Vacuum style since it is easier to understand and manipulate.

Figure 3.6 shows one of the use cases where on artist attempted to embellish the duck cloud with eyes. The range where the rub-out takes effect is determined by the height and the thickness of 3D canvas. Similar to painting, users need to define their desired camera view and 3D canvas before erasing.

Table 3.1 highlights the differences between two erase styles. The Vacuum style requires a more complex control, which involves cooperations between both the stylus pen and keyboard while the Rub-out style works on the pen only.

Table 3.1: Differences between two erase styles.

	Vacuum	Rub-out
Affected Area	spherical region	cylindrical region
Required Coordinates	XYZ in 3D canvas	XY from viewing plane
Location Control	pen movement and pressure	pen movement only
Diameter Control	keys: '–' and '+'	pen pressure
Erase Trigger	key: 'space'	tip press

The Rub-out style also produces a smooth boundary in a cylindrical shape. This unrealistic regularity can be eased by repainting along the boundary: a more comprehensive solution is left for future work. Nevertheless, it is necessary to have the erase function in our cloud modelling system as it provides a basic shape management on an existing cloud data.

3.5.3 Miscellaneous Functions

In addition to the erase mode, we also provide some other important functions to improve user's interactions with our system.

Light Control

The Light mode changes the location and color of the light source by pen movement. In section 4.5.2, we explained that the shadow map generation involves a light transmittance calculation for all voxels in a light volume. With 256^3 voxels in the cloud volume, this light volume produces realistic soft shadow at the cost of only two frames per second.

In this mode, the light volume is downsized to 128^3 so as to increase the performance to 16 fps. This enables light interaction with the support of a 3D shadow map at low-fidelity, which improves the user understanding of the cloud structures through shadowing and scattering effects.

Once the light is set in this mode, a mode switch from the Light to the Camera leads to the shadow map being generated at the full 256^3 light volume in order to present the best visual quality in the Camera mode.

Menu UI

The Menu interface enables command execution through pen tapping. Users are able to pop up the menu interface either by pressing 'M' on the keyboard or the 'Upper Button' on the side of the stylus pen.

Figure 3.7: A Menu interface for commands execution through pen tapping. The colour of the central menus indicates which mode they belong to. Each mode only presents its related commands in the center.

Figure 3.7 shows an example of the menu interface in the Camera mode. The menus at the bottom row represent five key commands for mode switching in our system. The Undo and Redo commands are frequently triggered by the users. The central menus with the same orange colour indicate that they are commands grouped in the category of the Camera mode. This visually indicates what functions each mode has, which improves understanding of our system.

It is worth noting that most commands have a hot key indicated in the corner of their menu buttons. For instance, the five mode switching commands can also be executed by keys from '1' to '5' on the keyboard.

History Track

Undo and Redo are implemented to allow recovery from mistakes made by the users. We track the camera status, canvas status, light status and, most importantly, the cloud dataset, in a history buffer.
Each Undo and Redo operation requires about 0.5 second for compression and decompression of the 3D dataset using a lossless data compression library, Zlib [40]. This significantly reduces the memory usage, because a 64 MB 3D dataset can be compressed to as little as a few hundred kilobytes.

3.6 Summary

This chapter discusses the structure of our system from the perspective of both programmers and artists. We start with a brief overview of state-of-the-arts sketch-based modelling systems, which leads to a major mapping problem. A hypothesis is then proposed and assessed through the development of a low-fidelity prototype. Although the result reveals a number of new challenges, tackling the mapping problem from the artist's perspective is a promising approach. This prototype shows the structure of our cloud modelling system, a voxel-based data structure which is implemented for an intuitive user scenario. Both the functionality and the usability can be improved using this foundation.
To support the workflow outlined in Figure 3.4, the interface thus has four main functions: camera control, canvas adjustment, painting and erasing. Apart from these, we integrate an erase mode, a light control mode, a pen-based menu interface and a history track function into our cloud modelling system, which aims to enhance the usability of the cloud painting during the experiment.
In the subsequent chapters, we will discuss the techniques underpinning these functions in further detail.

Chapter 4

Cloud Visualisation

In the previous chapter we noted that the voxel-based data structure is the core of our cloud system. This leads to a vital module of our modelling system, a visualization module, which aims to render the 3D dataset correctly and efficiently. This chapter presents our volumetric cloud visualisation in three steps: a GPU-based volumetric rendering, GPU-based octree process for performance enhancement and a fast light scattering simulation with volumetric shadowing.

4.1 GPU-based Volume Rendering for Clouds

Visualizing complex 3D models with high resolution is fundamentally important in computer graphics. With rapid developments in computer graphics, modern graphics cards are capable of rendering millions of vertices in less than a second. However, using triangles to accurately represent the shape of amorphous objects such as smoke, clouds and fire simulations, remains difficult. Although many approaches [48, 111, 103] are able to represent a convincing volumetric cloud effect, these vertex-based approaches still have drawbacks such as poor control of the volume data. Nevertheless, the volumetric rendering technique enables the display of both the surface and details within the volume. Although volume rendering requires a large amount of data and many calculations, Engel *et al.* [36] conclude that modern GPUs are a suitable solution for real-time rendering with the help of unified shader units and general-purpose

processing capabilities.
In this section, we focus on GPU-based volumetric rendering, which comprises slice-based and ray casting volume rendering systems.

4.1.1 Slice-based Volume Rendering

Cabral *et al.* [12] reformulate the traditional slice-based volume rendering techniques into algorithms that suited to hardware texture mapping and summing frame buffer. While they achieve this performance enhancement on a high-end graphics workstation, Westermann and Ertl [113] introduce a multi-pass algorithm for isosurface rendering on a OpenGL supported graphics card. Meißner *et al.* [76] expand on this hardware-based method to support accurate light shading of semi-transparent objects. Nevertheless, multi-pass algorithms remains a the bottleneck on rendering performance for rasterization hardware. Due to the fast development in graphics cards, many consumer cards have evolved large video memories, which can be accessed much faster than system memory. GPUs have started to support advanced texture fetches and programmable shader units. Engel *et al.* [35] take advantage of this new GPU power and provide a texture-based volume rendering approach using consumer graphics cards. They scale the size of the dataset to fit the graphics memory using a 3D texture or a stack of 2D textures. During rendering, the dataset is sliced into layers in the rasterization stage. These slices are taken such that their normals are aligned with the view direction. Pixel Shaders then blend these slices together and resample them to provide a convincing volumetric appearance. This method is used in many flight simulation applications for 3D cloud rendering. Harris [48] uses this approach to simulate a large number of cumulus clouds. He also tackles the 'slicing' artifacts that occur when looking from inside a cloud, by accessing a pixel's depth value.
There are many drawbacks to slice-based volume rendering [57, 101]. Ikits *et al.* [57] identify three major issues. Firstly, slice-based volume rendering processes a large number of geometric primitives which leads to many pixel operations. This can cause a performance bottleneck in the rasterization stage of the pipeline. Secondly, since both the calculation complexity using the pixel shader and the large number of texture

fetches can affect performance significantly, profiling performance and adjusting the balance between these two is challenging. The third limitation relates to texture memory. Since the trilinear interpolation used in 3D texture lookups is at least four times more costly than the bilinear interpolation in 2D textures, the speed of texture access becomes critically impacted.

4.1.2 Ray Casting

Another important GPU-based volume rendering approach is ray casting. Instead of merging a stack of textures, all pixels in the rendering window directly emit rays into the volume data. The rays traverse the dataset and return a colour. Since dynamic branching and loop operations have become more efficient in pixel shaders, using a direct ray casting method now allows interactive performance and convincing quality. Krüger and Westermann [62] propose a multi-pass strategy to pre-compute the ray data before ray casting. The ray data consists of the beginning position and the length of all the marching rays in 3D texture space. To collect ray data, a normalized 3D box is initialized to represent the ray casting 3D volume. The GPU first renders this box twice, using different culling modes. Both the front face and the back face of the box are rasterized, so that all the pixel-related 3D coordinate locations can be accessed through a transformation of UV coordinates on the pixel shader. These coordinate locations are stored in two separate textures, the front face texture and the back face texture. The ray data can then be collected by subtracting the coordinate locations stored in these two textures.

The cost of a single 3D box rendering and two extra rendering passes is negligible, especially on modern GPUs. One important benefit of this approach is that the type of dataset can vary. The dataset can be static or dynamic, such as a smoke simulation. Crane *et al.* [18] and Zhou *et al.* [123] extend this approach with the support of complex light shading so as to enhance visual quality.

When a GPU's parallel power is combined with a multi-core CPU, the performance of volume rendering can be increased dramatically. Gobbetti *et al.* [42] introduce a

method to render a massive volumetric dataset in real time. They use an adaptive out-of-core technique to distribute the workload between the CPU and GPU. First, a large volume dataset is decomposed into small cuboid bricks and organized into an octree structure. The CPU works as an adaptive loader during rendering. It keeps filtering the empty bricks in the octree according to the view direction and asynchronously transfers the brick-related datasets to the GPU. The GPU then renders these non-empty bricks using an efficient stackless ray casting algorithm. Crassin *et al.* [19] propose a similar system which is able to render a volumetric dataset containing billions of voxels with even better performance. They use an N^3 tree with a brick structure which reduces the number of pointers by a factor of eight, so that the memory usage is more efficient. They also shift the different levels of details from the CPU to the GPU to increase both the performance and quality.

4.2 Our Volume Rendering Implementation

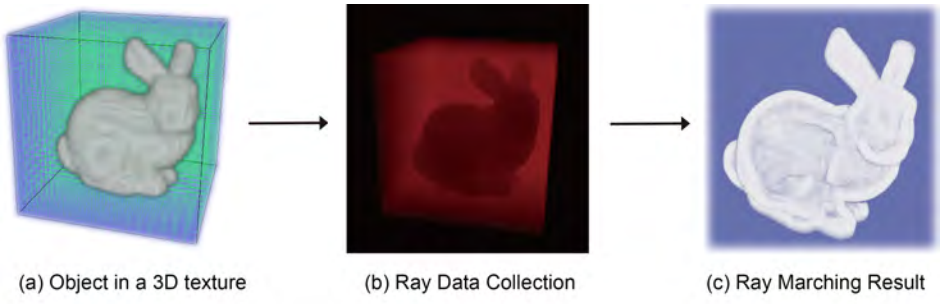


Figure 4.1: The workflow of our rendering system. (a) represents a raw 3D dataset as the input. (b) illustrates the collection of the ray data for volumetric rendering inside a box. (c) shows the final rendering result.

The GPU-based volume rendering we implemented is based on a methodology proposed by Krüger [62] and Crane *et al.* [18]. We therefore give a brief summary of their method in this section.

The flow of a common GPU-based volumetric renderer is illustrated in Figure 4.1. It starts with a volumetric dataset (Figure 4.1(a)) which is contained in a 3D texture. This can be considered as the input of the system. The rendered result (Figure 4.1(c))

is achieved using a GPU-based ray casting method. Before proceeding to the ray casting, there is a ray data collection stage (Figure 4.1(b)).
In this stage, the lengths and the directions of the rays are collected. These attributes are vital for the ray marching stage that follows, as they determine where the rays start, what directions they emit in and how far they march. The length that the ray is going to be cast is indicated by the intensity of red. Notice that significant ray casting occurs around empty regions in the volumetric box.

4.2.1 Ray Collection and Casting

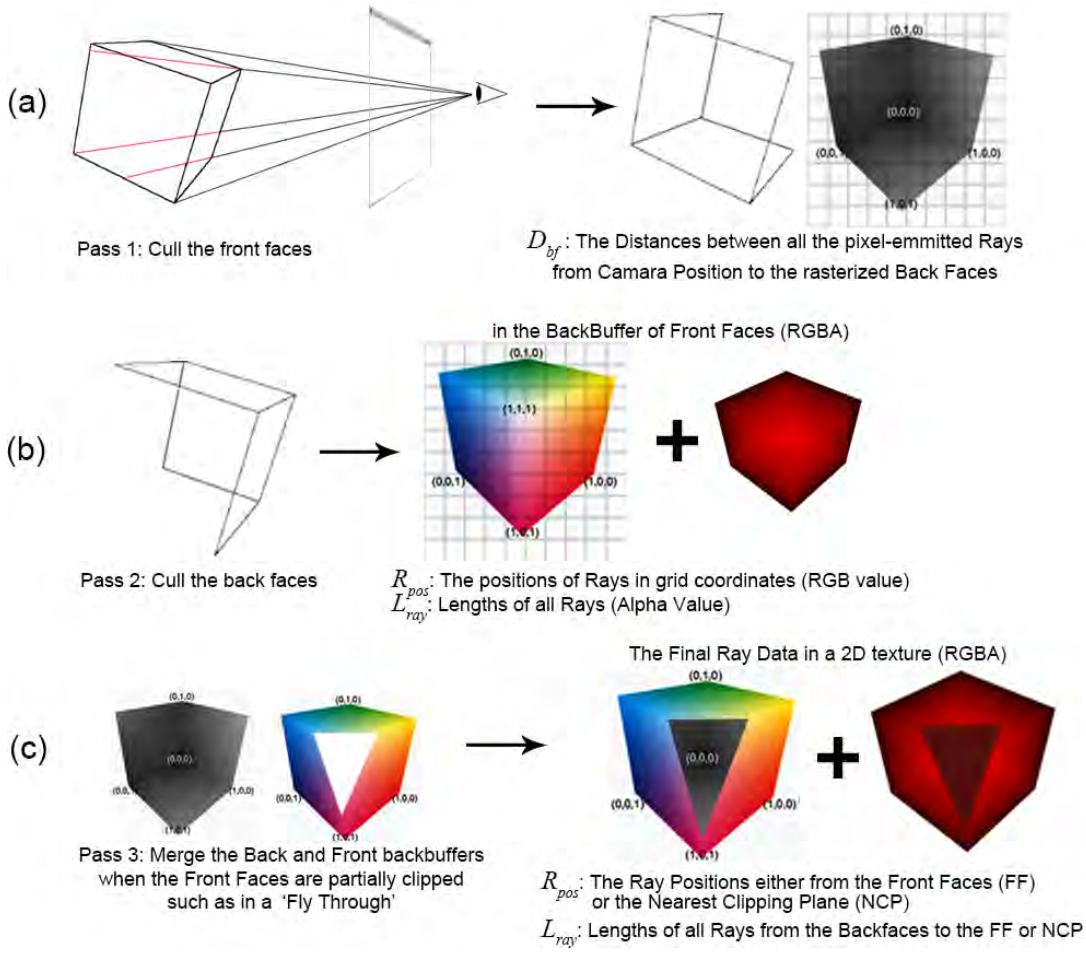


Figure 4.2: The GPU-based ray data is collected in three passes. Note the final ray data is collected in a 2D texture with values of R_{pos} and L_{ray} as shown in (c).

With help from the GPUs parallel power, Krüger *et al.* [62] successfully implement the entire workflow on the GPU with programmable shader units. One of the important contributions in his technique is the GPU-based ray data collection. Figure 4.2 shows the procedure for this data collection. The final data, as shown in the bottom right of the figure, is stored in a 2D texture. The texture follows the same resolution and format as the frame buffers of the rendering window. It contains all the 3D position values of the pixel-emitted rays in camera view space R_{pos}, which are stored in the RGB channels. It also holds the lengths of the rays L_{ray} in its Alpha channel, which is represented by the red intensity in the rightmost cube of Figure 4.2(c).
The point of this procedure is to leverage the parallel power of the programmable shader units for fast location and length determination of all the rays emitted in the cube.
First, the volumetric dataset is represented by a normalized 3D box with its front left bottom vertex situated at $O(0, 0, 0)$. This cube, as shown in Figure 4.2(a), is then rendered twice to distinct 2D textures. The red lines in figure 4.2(a) refer to the rays which traverse the 3D texture.

Algorithm 1 Ray data collection of the back faces (Pass 1).

```
1: //The CPU call for rendering back faces to texture TexBack;
2: DrawBox(TexBack, CullFront);
3: //The HLSL codes;
4: function VS_Back(float3 p : POS)
5:     SV_POS ← mul(float4(p, 1), WorldViewProjection);
6:     worldViewPos ← mul(float4(p, 1), WorldView).xyz;
7: end function
8: function PS_Back(float3 worldViewPos : TEX0)
9:     D_bf ← length(worldViewPos);
10:    return float4(0, −1, 0, D_bf);
11: end function
```

The first pass uses a clockwise culling mode, which culls the front faces and rasterizes the triangles in the back of the cube. Algorithm 1 lists the pseudo code of both the CPU call and the HLSL code for this rendering pass, which achieves four goals:

1. It culls the pixels that are not contributing to the volumetric rendering. This

significantly increases the performance.

2. The distance values, D_{bf}, from the camera position to the pixels of the back faces in the camera view coordinates, are stored in a 2D texture 'TexBack'.
3. The distance values, D_{bf}, will also be used in both pass 2 and 3. As the final ray lengths L_{ray} are achieved through the subtraction of the distance to back faces D_{bf} from the distance of the front faces D_{ff} or the nearest clipping plane D_{ncp}.

$$L_{ray} \quad = \quad D_{bf} - D_{ff}|D_{ncp}$$

4. In the case of a fly through part of the front faces are clipped during Pass 3, and the Green value is set to -1 so as to identify that the ray is hitting the back face directly from the clipped region. In this case, D_{ncp} is used for length calculations.

Algorithm 2 Ray data collection of the front faces (Pass 2).

```
1: //The CPU call of rendering front faces to texture TexFront;
2: DrawBox(TexFront, CullBack, &TexBack);
3: //The HLSL codes;
4: function VS_FRONT(float3 pos : POSITION)
5:     SV_POS ← mul(float4(pos, 1), WorldViewProjection);
6:     posInGrid ← pos;
7:     worldViewPos ← mul(float4(pos, 1), WorldView).xyz;
8: end function
9: function PS_FRONT(float3 posInGrid : POS, float3 worldViewPos : TEX0)
10:    R_pos ← posInGrid; D_ff ← length(worldViewPos);
11:    L_ray ← Sample(TexBack) − D_ff;
12:    return float4(R_pos, L_ray);
13: end function
```

The second pass, as shown in Algorithm 2, uses a counter-clockwise culling mode, which renders the front faces only. This pass desires not only the rays' distance values from the camera position to the front faces but also the rays' initial positions from the front faces (R_{pos}). All these values can be fitted into a 128bit RGBA pixel format, which can be stored in a single new 2D texture, TexFront.

Therefore, the front faces indicate where the rays start tracing through 3D texture

space and the back faces indicate where the rays terminate. The length of each ray can be retrieved by subtracting the distance value for the front faces from the corresponding back face one values. Although the length value L_{ray} is stored in the Alpha channel of the texture, it is represented by the red cubes in Figure 4.2(b)(c) for easy recognition.

However, there is a situation that occurs quite often during a fly through. The front faces are partially clipped when the 3D cube is viewed closely or from the inside, which leaves these pixels empty after the rendering pipeline. The correct process is to identify this empty region from the front faces and calculate the ray positions R_{pos} from the nearest clipping plane. Thus, another pass is implemented to merge both the back faces and the front faces. In this pass, both the TexFront and TexBack from the previous passes are loaded in the pixel shader. The pixel shader identifies the clipping region from the green value reading in TexBack, as we have set the back faces with value -1 in the green channel in Pass 1. The pseudo code is listed below.

Algorithm 3 Ray data collection of a fly through situation (Pass 3).

```
1: //The CPU call of rendering a quad to TexRayData;
2: DrawQuad(TexRayData, &TexBack, &TexFront);
3: //The HLSL codes;
4: function PS_PASS3(float3 posInGrid : POS, float3 worldViewPos : TEX0)
5:     rayData ← Sample(TexBack);
6:     if (rayData.g == −1) then
7:         rayData.xyz ← posInGrid;
8:         D_ncp ← DistanceToNearPlane();
9:         rayData.w ← rayData.w − D_ncp;
10:    else
11:        rayData ← Sample(TexFront);
12:    end if
13:    return rayData;
14: end function
```

After the ray data is collected, rays from all the pixels are cast in a parallel manner. They follow the direction of the camera's view and penetrate the 3D texture. The 3D texture is sampled at each step of the ray traversal. This means each pixel, which emits one ray, has multiple samples along its ray. These sample values are blended in

a front-to-back order:

$$\begin{aligned} C_{dst} &= C_{dst} + (1 - \alpha_{dst})\alpha_{src}C_{src} \\ \alpha_{dst} &= \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \end{aligned}$$

Here, the C_{dst} and C_{src} are the colour values in the current frame buffer and incoming pixel buffer respectively, α refers to the opacity value.

Different pixels have various sampling steps. The number of steps depends on the L_{ray} value from the ray data texture. The ray tracing terminates once it reaches the length of the ray data. It can also be stopped when its accumulated opacity reaches a cutoff value, such as the value of 0.9 in line 6 of Algorithm 4.

Algorithm 4 The pseudo code for ray casting.

```
1: R_pos ← Sample(texRayData).rgb; L_ray ← Sample(texRayData).a;
2: v3Step ← Normalize(R_pos − v3EyeOnGrid)/v3GridSize;
3: for (s ← 0; s < L_ray; s++) do
4:     fAlpha ← Sample(tex3Ddata, R_pos);
5:     v4Color ← ColorBlend(fAlpha);
6:     if v4Color.a >= 0.9 then
7:         break;
8:     end if
9:     R_pos += v3Step;
10: end for
11: return v4Color;
```

Figure 4.3 shows four simple letters with a semi-transparency effect we constructed for a rendering test. These four letters are three dimensional with the depth parallel to the z axis. They have different thickness which leads to various termination distance for the marching rays. Figure 4.3(c) offers a view that flies through the letter 'e'. Although the current ray casting method is based on a simple alpha blending approach, this fly-through produces a smooth interpolation of the semi-transparent color values between neighbouring voxels.

Nevertheless, one major problem with this system is the low frame rates even without including other fancy features such as complex light illuminations. In particular, the performance drops significantly when rendering a 3D texture with a sparse dataset

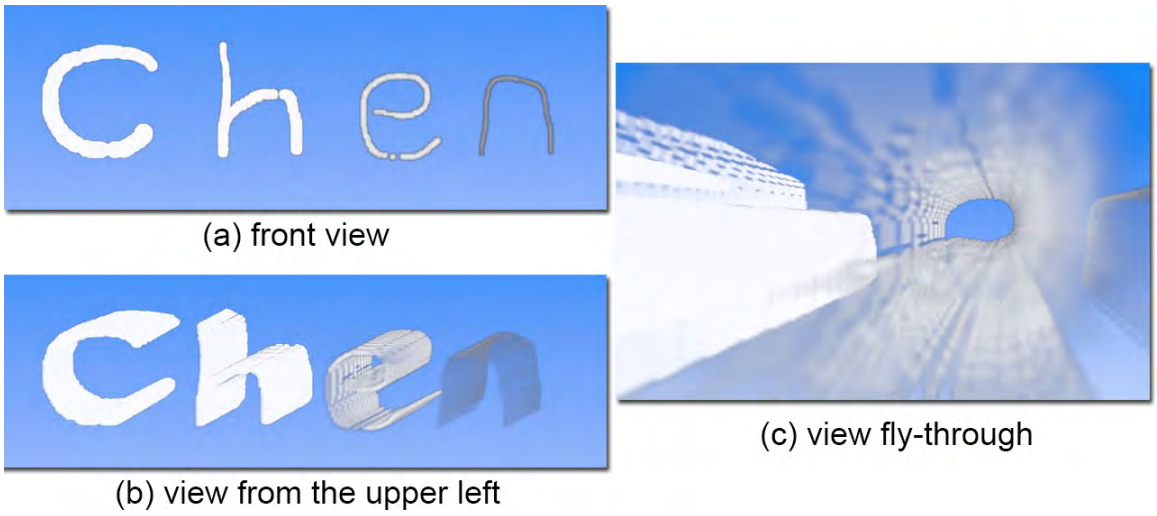


Figure 4.3: The volumetric rendering results of four 3D letters from different views. Note the transparency that simulates one of the basic cloud features.

due to the ray marching through the massive empty regions. On a laptop with an AMD hd5650M graphics card, an empty 3D dataset of 256^3 size is rendered at 12fps on average in a window of 1280×768 pixels. This performance needs to be improved as it can severely hamper the user experience for interactive modelling purposes.

4.3 Profiling Ray Casting

Volumetric rendering is expensive and time-consuming. It requires emitting rays for all the associated pixels and sampling a 3D texture multiple times for each ray. The total number of ray samples can easily be in the millions. This is not a trivial task even when one is working in parallel with GPUs. Nevertheless the sheer quantity of rays does provide an indication for a performance bottleneck.

To find out how the number of ray samples affects performance, we need to collect sampling statistics during real-time rendering so as to track the performance variation from multiple interactive viewing directions. As each pixel emits a number of ray samples during the ray casting, the statistics involve:

1. The total number of ray samples of all the pixels in a rendering frame (N_t);
2. The maximum ray samples among the pixels of the rendering frame (N_m);

3. The average ray samples over these pixels (N_a);
4. The histogram for the distribution of the ray samples of all the pixels (HSG).

The conventional way of computing these statistics is to use the CPU. During the ray casting process, an independent draw call is executed to draw a 2D texture which has the same resolution as the viewport. Each texel, corresponding to a screen pixel, contains the number of ray samples in the pixel's ray-tracing. This 2D texture is then sent to the CPU for computing the statistics, constructing the histogram and eventually sending the data back to GPU for histogram drawing. This process is time-consuming since it involves CPU-GPU memory transfer. This can cause a new bottleneck in real-time rendering and severely affects the original bottleneck detection. What we want is to find a quick method of computing statistics for the 3D texture sampling in real-time, which does not dramatically affect the performance of volumetric rendering. The new features, Unordered Access View (UAV) and Compute Shader (CS) in DirectX 11, enable us to compute the statistics and generate the histogram entirely on the GPU side.

4.3.1 DirectX 11 Features

DirectX 11 is Microsoft's new Application Programming Interfaces (API) for multimedia applications, designed specifically for games and graphics programming. It was released on 22 October 2009 together with the new operation system, Windows 7. There are five major new features in DirectX 11: Tessellation, Multithreading, Dynamic shader linkage, Compute Shader (CS) and Unordered Access View (UAV). CS and UAV are the two new features used in our statistics gathering.

Compute Shader (CS)

Compute Shader, which also refers to Direct Compute (DC), is for General-Purpose computing on Graphics Processing Units (GPGPU).
For general-purpose computing, when using the Pixel Shader (PS), the input dataset has to be adjusted to fit into a 2D texture. A rectangle must be drawn to cover the entire texture so that all the data can be processed. The output data must also

be adjusted to fit the size of the rendered texture. Thus, many of the procedures, which have to be executed when using pixel shader, are irrelevant to the actual data-processing.
Unlike the Vertex Shader and Pixel Shader, which are executed sequentially in the graphics pipeline, CS is a separate stage which can be executed independently. The input to CS can be any type of dataset and its output can target various data structures. A number of threads on the GPU are dispatched to process the dataset simultaneously. This is more efficient for general purpose computing than using the pixel shader.

Unordered Access View (UAV)

When PS outputs the processed dataset to a 2D texture, the sequence for drawing texels becomes unmanageable due to the synchronization barrier in parallel processing. It is also difficult to output a value to a specific location even in the same 2D texture. Furthermore shader programs have difficulty using arrays since dynamic indexing is not well supported in HLSL.
The new Unordered Access View feature is used to solve these problems. The UAV offers a new type of dataset in the High Level Shading Language (HLSL), the Unordered Access Buffer (UAB). This buffer can be read and written simultaneously by multiple threads. An added advantage is that this will not cause memory conflicts while using atomic functions.
The UAV can be used in both the Pixel Shader and Compute Shader. With the help of the API call, 'OMSetRenderTargetsAndUnorderedAccessViews', the Pixel Shader is able to output different datasets to two different places within the same draw call. While the pixel related dataset is being processed and rendered to the 2D texture, other temporary local variables can also be simultaneously written to an unordered access buffer. This buffer can later be used in the Compute Shader for general purpose computing.
Using these two features, most of the calculations related to various datasets can be processed entirely on the GPU. This reduces the number of CPU-GPU memory transfers, which leads to a performance enhancement.

4.3.2 Sampling Statistics in Ray Tracing

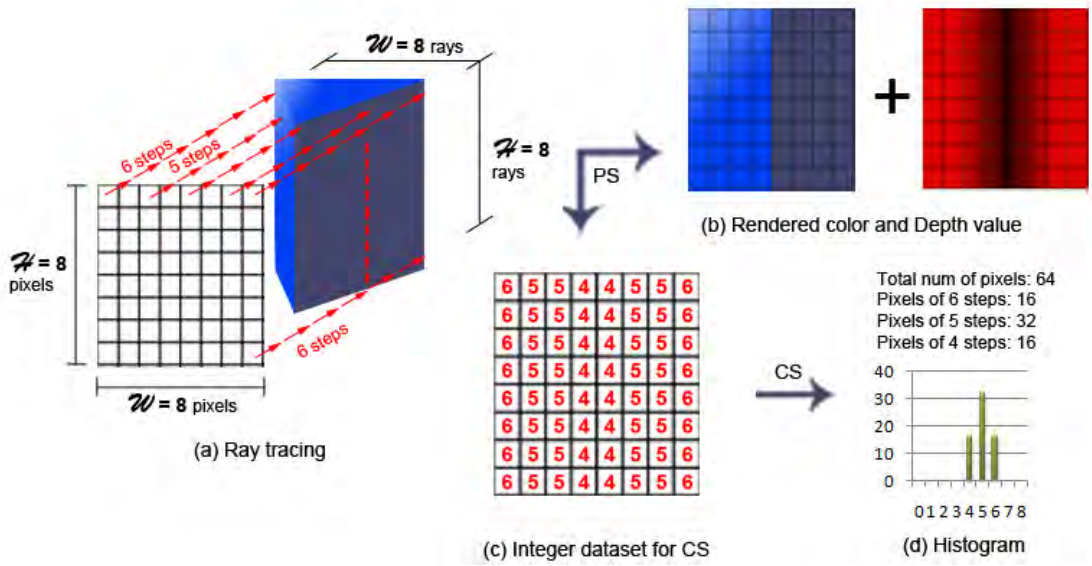


Figure 4.4: Statistics for 3D texture sampling in ray tracing.

Figure 4.4 shows a workflow, using DirectX 11 to process both volumetric rendering and statistical analysis. Figure 4.4(a) represents the stage of ray casting in volumetric rendering. This stage occurs directly after the ray data are collected, which is illustrated in Figure 4.2. Assume an object in the 3D texture is in the shape of a triangular prism. It is located in a direction as shown in Figure 4.4(a). In this case, a view of 8×8 pixels can cover the entire object. Each pixel in this view shoots a ray into the scene. Its direction is based on the result of ray data collection. The width (W) and height (H) are both equal to 8. This means there are 64 rays: 1 per pixel. Rays start marching from the boundary of the 3D texture. Each step samples the value at the corresponding texel in the 3D texture. The red arrows in Figure 4.4(a) represent the number of steps the relevant pixels take to reach the object.

Conventionally, a Pixel Shader can only output five values for each pixel, 4 for the RGBA value and 1 for the depth value, as shown in Figure 4.4(b). A new integer dataset that records the number of steps, as shown in Figure 4.4(c), requires another draw call for output. Fortunately, as already mentioned, a new method is exposed under DirectX 11: 'OMSetRenderTargetsAndUnorderedAccessViews'. This allows two

datasets to be outputted in the same draw call. Both of them are located in GPU's memory which can be easily used later for GPU-based analysis.
The dataset in Figure 4.4(c) is then directly bound as an input to the Compute Shader for statistical computation. This dataset could be in the form of a 2D texture, a 3D texture or a structured data buffer.

4.3.3 Histogram Construction

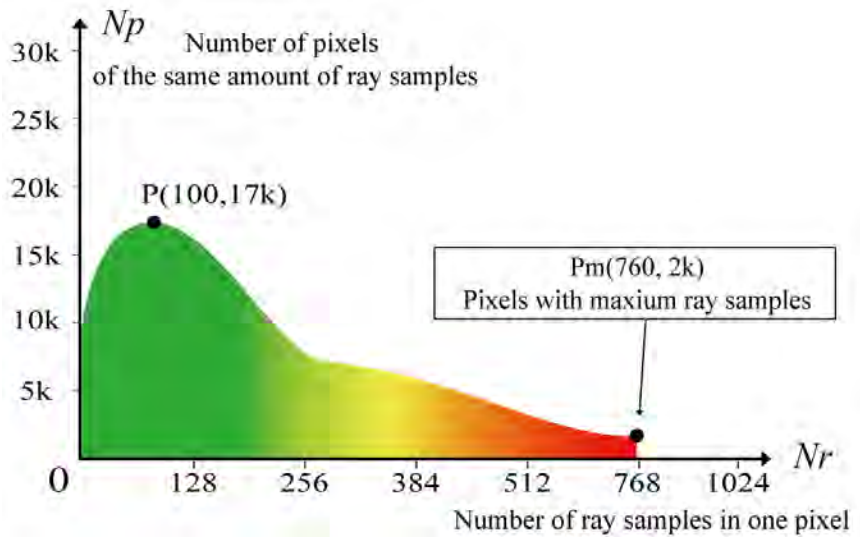


Figure 4.5: Histogram design for ray casting profiling.

As mentioned in the section 4.3, the aim of the statistics is to find the bottleneck of ray casting performance. Besides the statistical values of N_t, N_m and N_a, it is useful to have a histogram that represents the distribution of all the pixels in relation to their ray samples. Figure 4.5 shows the design of our histogram.
In this histogram, the x axis refers to the number of ray samples in one pixel (N_r) and the y axis indicates the number pixels of the same amount of ray samples (N_p). For example, the point $P(100, 17k)$ shows there are 17 thousand pixels that march 100 steps for sampling in ray casting. The point $P_m(760, 2k)$ lies in the top right corner of the histogram, which indicates the longest distance a ray has marched in this rendering frame (N_m) is 760 steps. The number of the pixels that sample the

same steps is 2 thousand. The distribution of this example histogram also denotes that most ray casting pixels march less than 256 steps. The number of pixels reaching the maximum ray samples is relatively small, which usually represents the rays that trace through the empty region in real application. Therefore, the ideal distribution we expect for good performance of the ray casting has three features:

1. Curve envelope should decay quickly as the number of ray samples increases.
2. The curve decreases along the x axis, which represents the number of ray samples per pixel.
3. The maximum Nr is small.

This example histogram represents such a good performance for the ray casting. In order to construct this histogram, an 1D data array of size 1024 is required that each element stores the Np value. The CS stage, as shown from Figure 4.4(c) to Figure 4.4(d), produces this 1D data array from a 2D dataset using a GPU-based data reduction operation [46, 121].
In real profiling such as the Figure 4.6, the performance of the volume rendering dropped by 7–10 fps and stabilized at around 18 fps, which is about a 30% loss compared to the original 25 fps. However, the performance barely reaches 3 FPS when transferring the statistics computation and histogram drawing to the CPU.

4.3.4 Profiling Result

Figure 4.6 shows the use of statistics computation in the real application. This application implements the GPU-based ray casting through a 3D texture which contains a cross-like object. The left image shows the depth values of the object. The right is the histogram of the statistical computation. The total number of samples N_t exceeds 154 million, which slows the rendering speed significantly. The max samples on one pixel N_m is 541, which is indicated by the pink vertical line in the histogram. In this histogram, the red part of the distribution is shaped with substantial red peaks, which indicates most pixels that individually sample the texture more than 512 times merely on the empty regions as indicated by the large red area in Figure 4.6(a). Thus, reducing the amount of ray tracing for these pixels can significantly increase

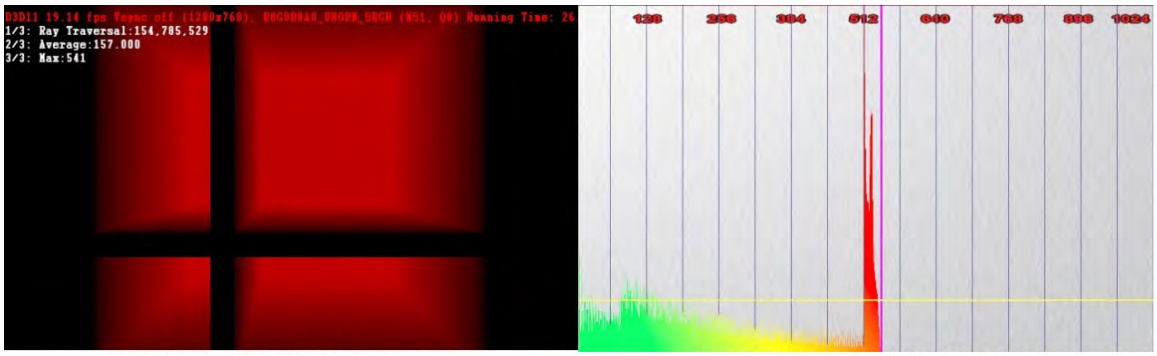


Figure 4.6: Statistics computation in real application. (a) The ray casting of a cross-like object which contains a large empty area (crimson region). (b) The corresponding histogram running in real-time. No explicit axis information due to the performance consideration but following the same structure as the Figure 4.5.

the rendering speed.

There are various space data structures that can be used for this kind of performance optimization, such as BSP trees, Octrees and Kd-trees, etc. However, most of these structures require pre-computation when generating the data. This process is also time-consuming and not suitable for scenes with dynamic content in real-time, such as in our case.

Crane *et al.* [18] use a straightforward method to reduce the number of samples. A smaller viewport, which is 64 times smaller than the original viewport, is used firstly for Empty Space Skipping (ESS). The procedures for ESS are mainly the same as for volumetric rendering, such as Ray Data Collection and Ray Casting. The only difference is in the decision to terminate the sampling in ray tracing. In volumetric rendering, rays often sample multiple times when they go through a semi-transparent object. This does not apply to ESS, ray tracing terminates once the sampling colour is non-zero. Apart from early ray termination, the total number of the ray samples is 64 times smaller than the original viewport. Therefore, the computational cost of ESS generation is limited compare to the full-sized ray casting. When using the ESS to filter the rays in comprehensive ray-tracing, it can significantly increase the rendering speed.

Figure 4.7 shows the result of the volumetric rendering on the same cross-like object

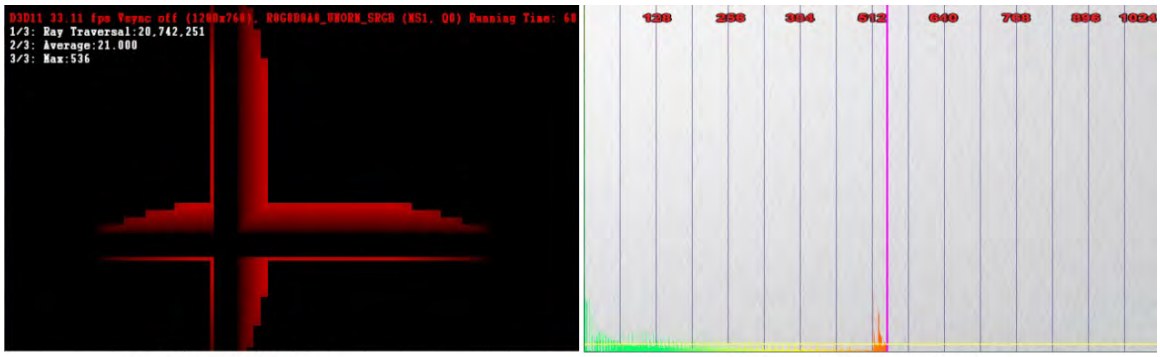


Figure 4.7: Volumetric Rendering with Empty Space Skipping

filtered by ESS. Figure 4.7(a) shows that only a small part of the total pixels are processed in the final full-sized ray casting. The peaked red lines in the histogram of Figure 4.6 disappear. The total number of samples drops from 154 million to 20 million, and the rendering speed is more than 30 FPS, representing an increase of 50 percent.

This empty space skipping method is widely used especially in rendering of volumetric smoke, clouds and fire. Its main advantage is that the 3D texture is used directly without any form of additional pre-processing, such as tree construction. Nevertheless, this method has limitations:

1. By scaling down the rendering window, detail may be lost. This can cause rays to miss data during the detail rendering stage.
2. Performance is view dependant. Although the coarse rendering costs much less than the full rendering, it still requires millions of traversal steps. In a situation where most of the viewing area is non-empty, the performance increase is limited due to the extra costs for ESS texture generation.

This leads us to further explore the ESS techniques. The next section discusses a new method we propose for lossless empty space skipping.

4.4 GPU-based Octree Generation and Traversal

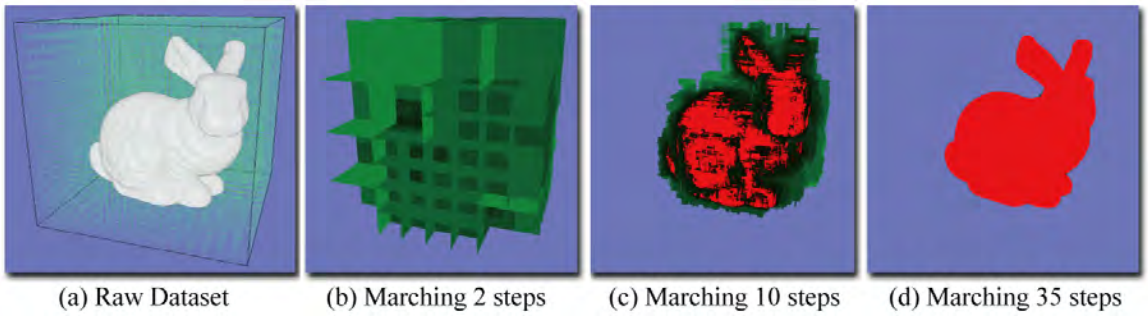


Figure 4.8: A full GPU-based Octree march through the dataset. (a) represents a raw 3D dataset contained in a 3D texture. This dataset is then transformed into an Octree structure, which is traversed by real-time Octree ray marching. (b)(c)(d) represent marching through the dataset after 2, 10 and 35 steps. While 2 steps just enters the Octree structure, 35 steps accomplishes detection of the full bunny in a 256^3 dataset.

Empty space skipping is enormously important in the field of computer graphics as it significantly decreases the calculation costs for various forms of collision detection. In the field of volumetric rendering, it reduces the workload of ray tracing for each rendered frame. Traditional empty space skipping, which involves space partitioning structures, is processed mainly on the CPU because the architecture of the CPU is suited to the large number of branch predictions required during tree parsing and traversal. This tree construction is normally time-consuming and generally takes place before the rendering stage and only works on a static dataset. Any change of the dataset requires a reconstruction of the tree structure, which stalls the rendering process.

Using a GPU to generate and traverse a tree structure is challenging. The architecture of the GPU makes it difficult to efficiently parallelize branching operations. However, the new DirectX 11, shader unit and Compute Shader, enables a way of implementing empty space skipping during volumetric rendering.

Figure 4.8 demonstrates the results of our completely GPU-based Octree generation and marching technique. This method enables fast empty space detection during GPU-based volumetric rendering. We define a new stackless Octree structure suited to GPU-based structure generation and marching. With the help of the Compute

Shader, different algorithms and mechanisms are implemented and verified for Octree generation. Our Octree marching is designed specifically for the new Pixel Shader. The workflow of our rendering system is illustrated in Figure 4.9. Before a full-sized ray casting occurs, we use this GPU-based Octree process to generate an empty space mask. This mask, which eliminates the 'empty' pixels, is used during the GPU-based ray casting stage to speed up volumetric rendering performance. The results improve the performance of empty space skipping in most cases. Furthermore, a dataset can be changed and rendered, together with the related Octree structure, at interactive speeds.

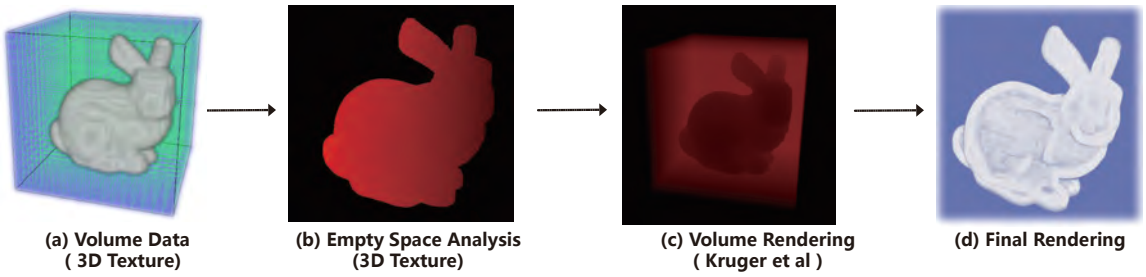


Figure 4.9: The workflow of our rendering system. (a) The input is a raw 3D dataset. (b) An empty space skipping region is generated by GPU-based Octree marching. This is used together with GPU-based ray casting (c), so that the final rendering (d) can be sped-up significantly.

The main novelties in this technique are:

1. Octree construction and traversal are executed completely on GPU;
2. A stackless Octree suited to the GPU is defined;
3. A fast Octree generation method using Mipmapping and Compute Shader is presented, and
4. A fast Octree traversal method on the GPU is developed.

In next section, we present a brief review of various space partitioning systems used for the volumetric rendering.

4.4.1 Background of Space Partitioning Systems

One of the major performance bottlenecks in ray casting is the time spent on traversing large areas of empty space in volume datasets. Efficiently determining how to skip the empty-space is important for both slice-based and ray casting systems.

The use of space partitioning systems, such as the BSP, Octree and Kd-tree, is a common approach to tackling the problem of skipping empty space in a fixed dataset. For texture-based systems, Li *et al.* [67] propose a method to partition the volume into sub-volumes with similar properties. These sub-volumes are then reorganized and stored in a BSP structure. This BSP structure is used for empty block skipping and occlusion culling, which can increase the performance significantly.

For ray casting a static dataset, kd-trees is also an efficient acceleration structure. Ernst *et al.* [37] present a stack for kd-tree traversal using the parallel power of the GPU. Foley and Sugerman [39] implement two stackless kd-tree traversal algorithms on a GPU, namely kd-restart and kd-backtrack. Popov *et al.* [93] introduce an efficient stackless kd-tree algorithm, which boosts the GPU ray tracing performance beyond CPU-based ones. Lefebvre *et al.* [50] propose a method to encode the octree structure in a texture referred to as an indirection pool. This reduces the memory usage and also exploits the modern GPU's parallel power for texture-based Octree traversal, since the pixel shader is good at accessing arbitrary textures.

However, most of the 3D datasets are static in the aforementioned systems. Before the start of volume rendering, the source 3D datasets, especially large ones, have to be restructured as an Octree. The datasets are subdivided into bricks. Then, the empty bricks need to be filtered. This dataset pre-processing can take hours, which does not match the interactive requirement of dynamic systems. For example, Gobbetti *et al.* [42] took 95 minutes to pre-process a 4.1GB dataset.

Our rendering framework is based on Krüger *et al.*'s [62] GPU-based ray tracing mechanism, since the data structure is flexible. We introduce a new hash system for GPU-based Octree construction. This octree construction allows dynamic 3D datasets to be processed at interactive speeds. Consequently, the volume rendering can take advantage of the Octree structure for a large performance increase. This hash system uses Compute Shaders, which are supported by all DirectX 11 graphics cards.

Octree Hash Table

Our dataset is encoded using a GPU-based octree structure. Most prevailing Octree structures used in this field fall into two categories: pointer-based and hashed.
The pointer-based Octree stores the addresses of the pointers in the nodes. For a complete tree, each node has eight pointers which address its eight children. It also has a pointer which points to the data if the node is a leaf. In our implementation, there are many extra pointers added to each node, such as a pointer to its parent node or a pointer to a temporary stack in the memory to increase the traversal speed. These pointer-based Octree structures are better suited to CPU traversal implementation due to the frequent branching operations and direct memory access.
The hashed Octree is another type of Octree representation. It stores the nodes in a hash table. The nodes can be accessed directly without using explicit pointers. The information of each node is encoded into a key, which can be used to identify and locate the related node from the coordinates of the node in space. There are many efficient definitions of such keys [108].
The Morton code [80] is a common coding system for hashed Octrees. An interleaving technique is introduced to efficiently locate a cell's depth in the tree hierarchy based on the cell's coordinate position. This technique can be accelerated by integer dilation [107], which converts the coordinate's values into integers first and interleaves the bits between integer values from different axes so as to get corresponding tree depth.
In next section, we discuss a novel key generation and coding system. It is similar in concept to interleaving the bits to form the key but better tailored to graphics hardware.

4.4.2 Octree-based Empty Space Skipping

In this section, we explain how our GPU-based Octree generation and traversal is implemented to improve the empty space skipping. The empty space skipping is divided into three stages:

1. Stackless Octree Generation

2. Homogeneous Region Generation
3. Octree Ray Marching

The first two stages (Stackless Octree Generation and Homogeneous Region Generation) only occur once for a particular static dataset before real-time rendering takes place. The format of the raw 3D dataset is that of a Microsoft DirectDraw Surface (DDS), with a size of 256^3. During frame rendering, rays first march through an Octree-structured 3D texture. In this way, the empty space inside the 3D dataset can be cut out or minimized before the later ray casting stage.

Stackless Octree Generation

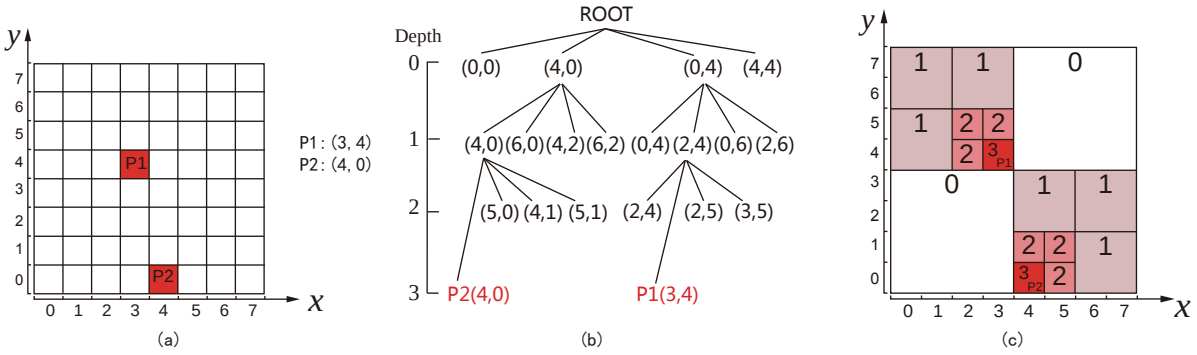


Figure 4.10: 2D volumetric dataset (a) with two nonempty cells P1 and P2. (b) represents its tree hierarchy with the deepest values for P1 and P2. (c) illustrates the stackless Quadtree result.

There are two requirements for our tree generation and traversal system. It must be fast enough to be interactive. Tree generation occurs frequently due to changes in the 3D dataset. We chose a stackless Octree structure because:

1. The Octree structure suits 3D textures, which are suited to GPU implementation.
2. A stackless tree structure can prevent a large number of branching operations, which is again important for GPU suitability.

Since our stackless Octree structure is only used to detect empty space, the key data required is an identification of nonempty cells in the 3D dataset. Figure 4.10 introduces the structure of a stackless Quadtree, which has the same principle as our

Octree generation. Figure 4.10(a) represents the input and Figure 4.10(c) represents the output.

Figure 4.10(a) shows a point dataset in an 8×8 2D regular volume. P1 and P2 are the cells containing data while the other cells are empty. Figure 4.10(b) illustrates the hierarchy of the corresponding 8×8 Quadtree. The nonempty cells, P1 and P2 are leaves at the deepest level. These depth values are used to generate the output dataset, as shown in Figure 4.10(c). Each node indicates the entry coordinates of the region that corresponding to the depth of the Octree.

Figure 4.10(c) shows the 2D texture-based stackless Quadtree corresponding to the dataset in Figure 4.10(a). All the cells contain a value from 0 to 3 that refers to their depth in the Quadtree hierarchy. Thus, all the texels in this 2D texture have an integer value from 0 to 3. A value of 0 indicates that the relevant 4×4 region is empty. A value of 1 means the region of size 2×2 is empty; a value of 2 means the 1×1 cell is empty. Finally, a value of 3 means the relevant cell is occupied.

There are several methods to generate this stackless Octree from a raw 3D dataset. To begin with we present an algorithm that is implemented on the CPU. Then, a complete Compute Shader implementation is introduced to speed up the generation process. Ultimately, a mix between Mipmapping and a Compute Shader is adopted, which enhances the performance of the generation step to real-time levels.

Implementation using a relationship equation

As Figure 4.10(c) shows, the value V_{empty} of each empty cell is defined by its position with respect to the nonempty cells. In other words, the position, or the coordinate relationship C between the nonempty cell and the other empty cells is the key factor in the algorithm for value calculations.

$$C = C_{empty} \oplus C_{nonempty} \tag{4.1}$$

$$V_{empty} = \log_2 D - \lceil \log_2(1 + C_x|C_y|C_z) \rceil \tag{4.2}$$

C_{empty} in equation 4.1 refers to the coordinates of the empty cell currently being calculated. Its X, Y and Z coordinate values are bitwise XORed ($\oplus$) with the coordinate values of the nonempty cells, $C_{nonempty}$. V_{empty} is the final depth of the

empty cell in the Octree. D represent the resolution. For a 256^3 Octree structure, D in equation 4.2 would be 256.
For the Quadtree in Figure 4.10, we assume for the sake of example that the value in an empty cell (2,5), C_{empty}, needs to be calculated. The XORed result C, with its coordinates compared with the nonempty cell $C_{nonempty} = P1(3,4)$, is $C = (1,1)$.

$$C = C_{empty}(2,5) \oplus P1(3,4) = (2 \oplus 3, 5 \oplus 4) = (010 \oplus 011, 101 \oplus 100) = (1,1)$$

For an 8×8 quadtree, the equation 4.2 becomes

$$V_{empty} = 3 - \lceil \log_2(1 + 1|1) \rceil = 2,$$

which matches the cell in Figure 4.10(c).
These equations are suitable both for CPU and GPU generation. However, the time complexity is $O(n^2)$. The execution time increases substantially when the number of nonempty cells increases. Therefore, we propose a new method to increase the performance by merging the Mipmap technique and a Compute Shader.

Implementation using Mipmapping

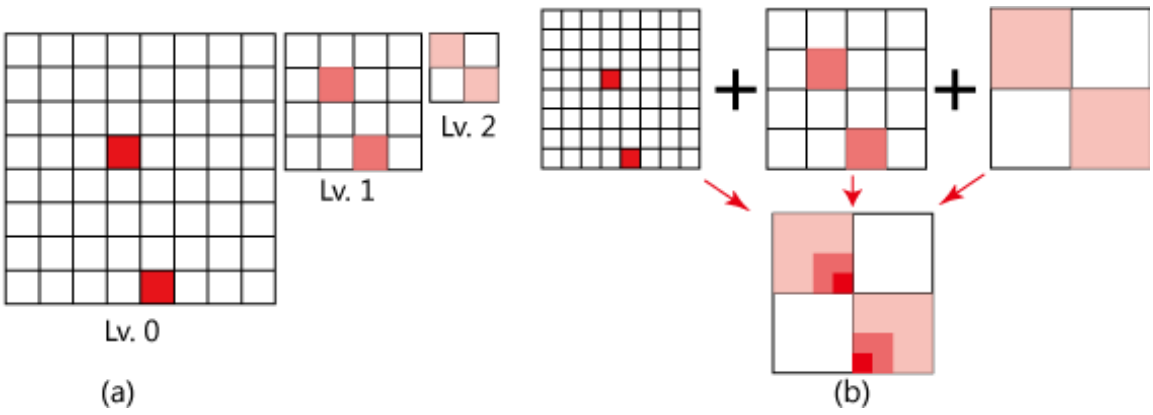


Figure 4.11: A series of Mipmaps at different levels. (a) uses hardware-filtered linear interpolation while (b) uses point interpolation and is implemented on the Compute Shader.

Mipmapping generally refers to a primary texture being sampled into a series of downsized images. Since DirectX 9, hardware-based Mipmap generation has been

supported by most graphics cards. These cards provide dedicated filter units to enhance the speed of texture sampling. This leads to high performance Mipmapping. During Mipmap generation, these dedicated filtering units are able to sample two adjacent texels for 1D textures, four adjacent texels for 2D textures and eight adjacent texels for 3D textures. Although these are the minimum sample numbers for linear interpolation in different texture dimensions, it is sufficient for our purposes. Tree generation now requires three stages:

1. Mipmap generation using linear interpolation
2. Enlarging all the downsized Mipmaps using point interpolation
3. Merging all the Mipmaps using the Compute Shader for 3D textures

For example, Figure 4.11(a) shows a Mipmap generated from the 2D dataset of Figure 4.10. Each level of the Mipmap represents a depth level in the Quadtree hierarchy. The level 0 Mipmap represents the deepest level of the tree while level 2 refers to the first level of the Quadtree. This Mipmapping uses linear interpolation supported by dedicated filters. Next, in Figure 4.11(b), the level 1 and level 2 Mipmaps are expanded to the same resolution as the level 0 Mipmap. This enlargement uses point interpolation to prevent the adjacent empty cells becoming nonempty. The last step is to merge these Mipmaps by combining the texel values from different Mipmap levels. Figure 4.10(c) is the result of Figure 4.11(b). This final merge can be implemented using a Pixel Shader for 2D textures.

However, for Octree generation in 3D textures, the Pixel Shader is not an ideal solution. In order to access texels inside a 3D texture, a 2D quad has to be drawn first. The GPU processors access the texels in parallel for only one depth of the 3D texture at a time. This requires extra loops to process all the depth levels of the 3D texture. Although optimization is possible, proxy 2D quad generation and a large number of loops cannot be avoided using the Pixel Shader. However, The Compute Shader can do a better job than the Pixel Shader. Unlike the pixel-based processing in the Pixel Shader, the Compute Shader is thread-based. These parallel threads are organized in three dimensions, with each thread identified by three integers (X, Y, Z). Once the thread ID is bound to the coordinates of a 3D texture, each thread refers to one texel

in this 3D texture. This enables the 3D texture to be accessed directly. No proxy geometry or nested loops over the depth are needed during access. The only task in each thread is to sample the associated texel values in all Mipmaps and to sum them.

4.4.3 Homogeneous Region Generation

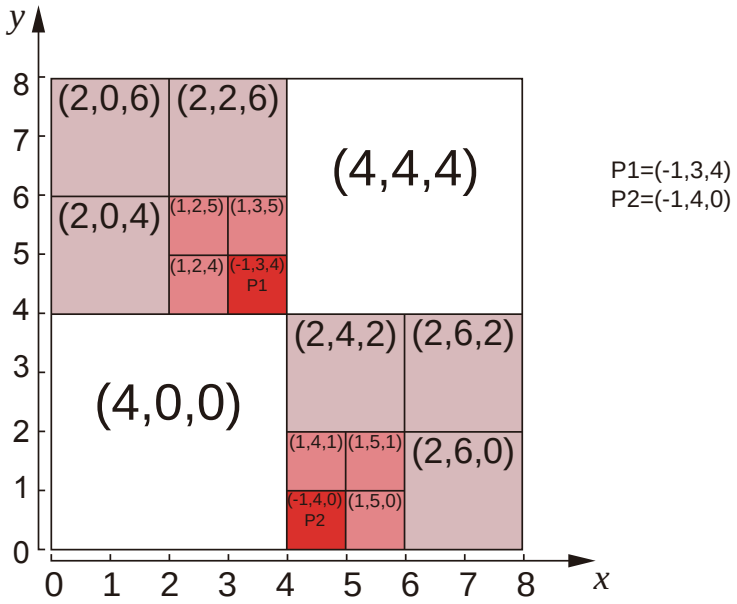


Figure 4.12: The homogeneous region data structure which is converted from the Figure 4.10(c) using a Compute Shader. Each set of three integers (iL,ix,iy) means the length of the homogeneous region and its XY position of the region.

After a stackless octree texture is generated in the previous stage, each texel only possesses a tree hierarchy depth value, which is not sufficient for ray marching through the Octree. A stage of homogeneous region generation is needed to create the information that is required for the later GPU-based Octree ray marching. Once a ray steps into the region, its sampling value should provide information such as

1. whether the texel is empty;
2. the size of the empty region, and
3. the bounds of the empty region.

The homogeneous region provides answers to the above questions. Each cell in a homogeneous region is described by three integer values (i3HV). These three integers are defined as (iL, ix, iy) in a 2D texture.

iL The length of the homogeneous region. -1 represents a nonempty cell.
ix The start x position of the region
iy The start y position of the region

Figure 4.12 shows the homogeneous region texture for the stackless Quadtree texture corresponding to Figure 4.10(c). Any cells in the white area in the top right contain the same i3HV value (4, 4, 4). This means if any ray steps into any cells in this area, the ray should skip a 4×4 square area which starts from (x=4, y=4) and ends at (x=8, y=8). By generalization, in a 3D texture, all the cells have a series of four integer values, i4HV (iL, ix, iy, iz).
With the help of the threadID feature of the Compute Shader, the conversion cost from the stackless tree into a homogeneous region is considered to be negligible from a pre-rendering perspective. The HLSL pseudo code for the homogeneous region conversion from a 256^3 stackless Octree texture in the Compute Shader is

for $i = 0 \rightarrow 2$ **do**
$iLength \leftarrow 8 \gg (i + 1)$
$int3Position = ((ThreadID \gg (3 - i)) \wedge 1) \cdot iLength$
end for

The variable i represent the depth value from the previous stackless Quadtree. $iLength$ and $int3Position$ variables are the components for the i4HV.

4.4.4 Octree Ray Marching

In this stage, the Octree ray marching starts and is processed in every rendered frame. It is executed in the Pixel Shader stage of the pipeline. To better describe the procedure, a 2D Quadtree texture is demonstrated again. Two rays R_a and R_b in Figure 4.13(a) are marched through the homogeneous region which is generated in the previous stage. Once R_a hits the white region at the point Sa1, it skips the

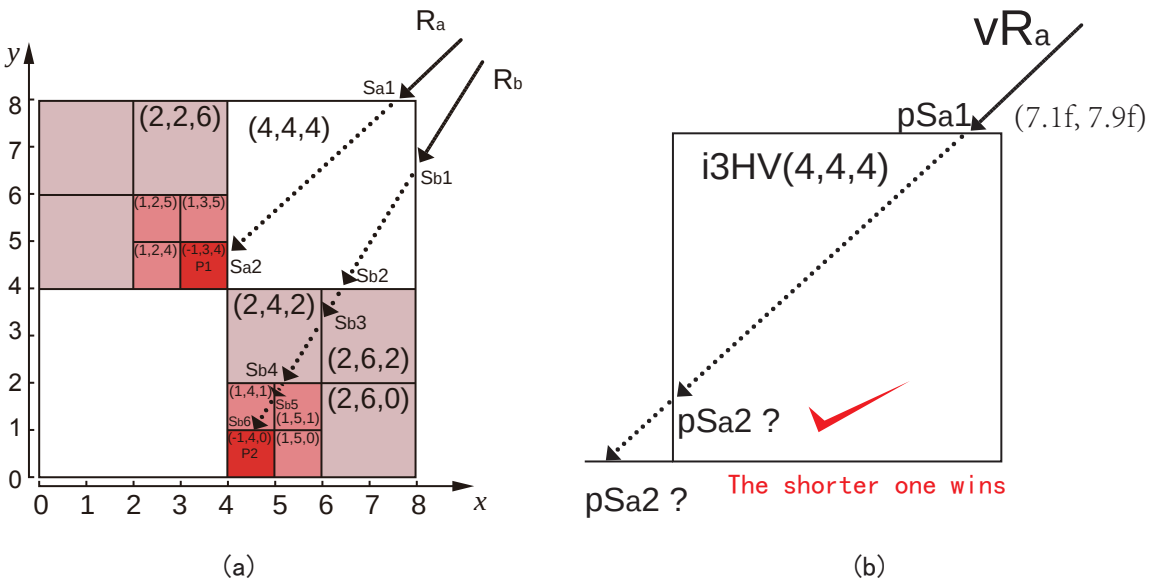


Figure 4.13: (a) Two rays, R_a and R_b are marching through the homogeneous region (b) The enlarged version of R_a marching through the white region

white region and resumes again at the point Sa2 to discover that a non-empty texel has been reached. In the case of R_b, there are five skips in total. Figure 4.13(b) is the enlarged version which depicts how the ray R_a marches through the white area in Figure 4.13(a). In Figure 4.13(b), vRa, pSa1, i3HV are given the following values before the skip occurs:

vRa The normalized vector direction of R_a
pSa1 The position of the hit point Sa1
i3HV The homogeneous value of the region

However, there are two possible intersection points for pSa2 before the ray starts skipping based on the above variables. The following 4 steps explain the strategy for ray skipping and how pSa2 is located correctly.

1. Once a ray hits the region, retrieve the region data (i3HV) by sampling the hit texel from the homogeneous region texture.
2. A ray emitted from pSa1 can hit all the other three edges of the cell based on its direction. We can filter out one edge based on the direction of the vRa. If

vRa points to the right, the left edge will never be hit. If the vRa points to the left, the right edge can be filtered. Thus, there remains only two possible hitting edges: the bottom edge and the one of the side edge.

3. Calculate the distances from pSa1 to both the possible intersection points.
4. pSa2 can be identified by choosing the one with the shorter distance.

Although this method causes branching in the Pixel Shader, the performance in dynamic branching has been significantly improved with Pixel Shader 5.0. With support from HLSL 5.0, the four steps mentioned above can be simplified into four short lines, where each line represents one step:

```
i3HV = Texture.Load(floor(pSa1),0);
float2 pSa2= vRa > 0 ? i3HV.r + i3HV.gb : i3HV.gb;
float2 fDis= (pSa2-pSa1)/vRa;
pSa2 = pSa1+min(fDis.x,fDis.y)·vRa;
```

4.4.5 Octree Implementation Results

ESS: Full vs Coarse Comparison of Data Integrity

As explained in Section 4.3.4, the coarse ESS method used by Crane *et al.* [18] can cause data loss. This is particularly noticeable for sparse datasets. Figure 4.14 shows a sparsely sampled 'ninja head' in a 256^3 volume. Both the full-sized ESS approach and our Octree traversal method are able to capture all the occupied cells. However, as shown in this example, a significant number of cells (100 in this example) are missed using the coarse ESS method. In the interest of maintaining accurate volume rendering, we focus only on the performance of full detail ESS methods.

Rendering Performance

Krüger et al [62] detect empty space in the 3D texture using a downsized pre-raycasting method. This risks losing data detail during the final, full-sized, ray casting. Although Octree datasets can easily be resampled to smaller dimensions, a full-sized rendering is better for determining performance differences. We thus conducted tests to compare

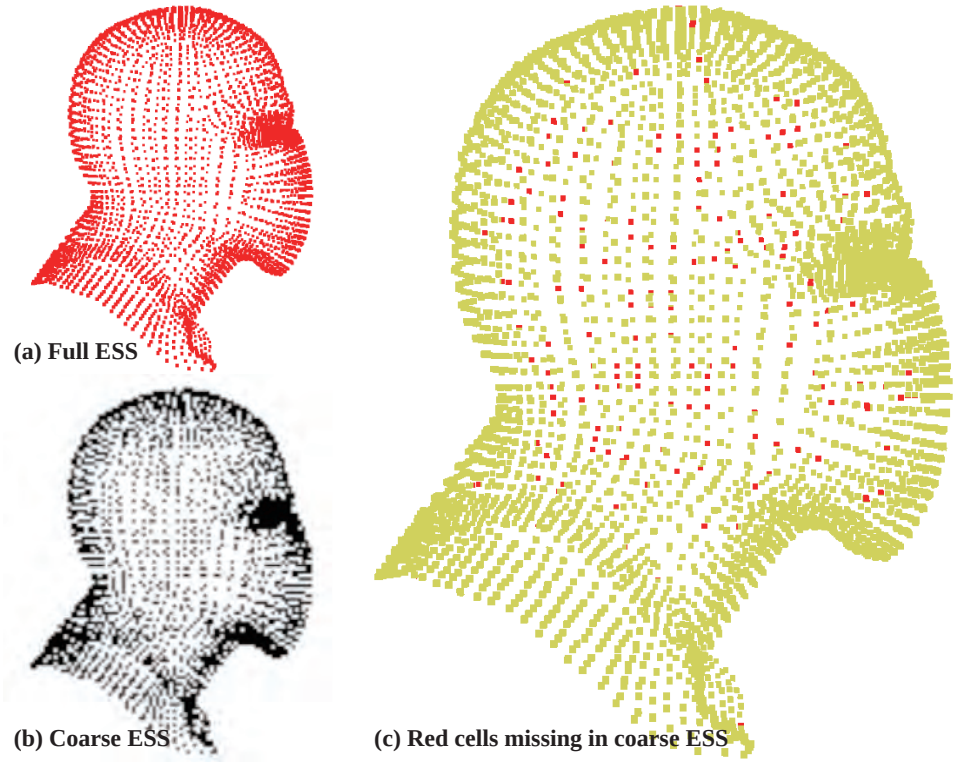


Figure 4.14: A comparison between full-sized ESS and coarse ESS. (a) is the full ESS result. (b) is the coarse ESS result. (c) merges both results so as to reveal the difference. The missing cells are highlighted in red colour.

the performance of full-sized ray casting on various datasets, full-sized empty space skipping as proposed by Krüger, and our GPU-based Octree marching.

We ran tests on fourteen 256^3 datasets of varying shape and cell occupancy. We focussed on sparse datasets, since we are interested in the performance of empty space skipping. Some of these datasets are provided by our 3D texture-based modelling system and others were converted from OBJ files found in the Stanford Data Archive. All datasets have transparency attributes to support alpha blending. An empty dataset is also provided as a baseline comparison. The results are presented in Figure 4.15. Three performance indicators were used:

FPS The key performance index, frames per second

Ratio The ratio of empty pixels/total rendering pixels.

All Steps The total number of the steps from all the marching rays

Name	Nonempty Cells in 256^3	No ESS			ESS			Octree		
		FPS	Ratio	All Steps	FPS	Ratio	All Steps	FPS	Ratio	All Steps
Empty	0	117.43	18.91%	124,364,002	121.15	18.91%	124,364,002	175.12	100.00%	0
Ninjiahead	4,455	117.61	18.91%	124,364,002	121.80	18.91%	111,916,363	129.32	91.63%	1,905,886
Hair	4,463	116.44	18.91%	124,364,002	121.47	18.91%	108,387,420	115.46	84.20%	4,007,641
Ateneam	7,283	116.65	18.91%	124,364,002	122.81	18.91%	110,297,046	126.95	93.03%	1,204,575
Venusm	9,026	116.09	18.91%	124,364,002	121.97	18.91%	118,441,102	139.85	96.70%	493,200
Two dots	9,814	117.41	18.91%	124,229,311	121.07	18.98%	124,119,342	163.99	99.69%	85,024
Building	13,952	117.26	18.91%	124,364,002	121.35	18.91%	111,690,429	128.93	89.88%	2,009,237
Four dots	28,328	116.77	18.91%	123,952,430	121.65	18.96%	123,253,498	158.59	98.28%	197,199
Three lines	29,410	117.87	18.91%	124,256,547	123.37	18.97%	118,647,717	144.81	96.25%	477,642
Bunny	34,219	117.54	18.91%	124,364,002	133.49	18.91%	99,760,797	144.36	85.50%	1,491,386
Cross	130,816	122.44	19.45%	115,945,529	126.45	19.61%	115,858,621	135.57	72.18%	3,891,313
Chen	252,166	118.01	18.91%	121,527,234	122.41	20.34%	119,529,368	147.47	94.21%	1,534,810
Africa	1,321,362	128.03	18.91%	93,496,594	138.54	27.01%	85,237,595	122.25	75.74%	4,419,009
Cloud	3,343,257	144.07	18.91%	61,897,719	155.50	38.40%	49,609,824	113.07	57.39%	7,555,813

Figure 4.15: Performance results from 14 datasets. Three aspects are compared: frames per seconds, ratio(empty/total pixels) and the total number of all ray steps.

Early ray-termination ensures that ESS-enabled ray casting shows marked gains over non-ESS ray casting. Although tree traversal is more expensive than ray sampling per step, our GPU-based Octree method marches much less frequently than the other methods, resulting in improved rendering performance in most of our tests. This performance depends mainly on the occupancy of the 3D texture. Our method gains almost 50% for sparse datasets, while the performance reduces on datasets which have high occupancy. More specifically, the larger the ratio value, the higher the frame rate since more rays are culled early.

Benefits in Cloud Modelling

Volume rendering in our cloud modelling system requires both a large number of rays and a large number of samples per ray. To reduce the computational complexity of this process, unnecessary ray sampling should be terminated quickly. This is the approach taken by empty-space skipping (ESS) methods. However, such methods do not necessarily perform well when the data is very sparse.

We have developed an GPU-based octree traversal technique based on hardware mip-mapping, and works in tandem with the DX11 Compute Shader to enable fast ray marching through volumetric data. This method provides an efficient rendering of sparse volumetric data. It improves our cloud modelling system in two respects:

1. **Higher Framerates with Better Accuracy** The cloud volume is first represented as an empty 3D canvas to the modellers. The modeller generates the 3D cloud by stroke painting, which usually produces a small 3D dataset. This empty or sparse dataset is well suited to our GPU-based octree traversal technique. Compare to the down-sized ESS approach, all the modelling details are preserved using our octree technique with at least 30% increase in rendering performance.
2. **Better User Experience** The improved performance leads to a better user experience when the modeller changes the camera positions and the viewing direction in the sky volume. The adjustment operation is smoother, particularly for the canvas adjustment. This provides users a precise control of the camera, canvas and the brush simulation.

4.5 Volumetric Cloud Illumination

Volumetric illumination is crucial to effective smoke and cloud rendering. It enhances rendering quality by providing a strong impression of depth and light. This area of rendering is already well explored.

4.5.1 Review of volumetric illumination

In the background chapter of this thesis, we presented several volumetric rendering methods focusing specifically on their suitability for cloud modelling. These methods provide impressive cloud visualisation with simple light implementation while achieving interactive performance. Although Harris [48] and Wang [111] propose different cloud modelling systems, their rendering modules use a similar method based on the work of Schaufler [102], in which the volumetric dataset is rendered using dynamically generated imposters. Their coarse light approximation provides simple cloud shading features mostly for cumulus clouds. Unfortunately, imposters can produce visible artifacts from certain viewpoint directions. Bouthors *et al.* [11] introduce a multiple anisotropic scattering method which accounts for the entire light path through the volume. However, this method only works on a Hypertexture, with the cloud surface represented by a mesh structure. Elek *et al.* [33] present continuously refreshing 3D photon mapping for cloud illumination. This produces a characteristic and realistic appearance for a single cumulus cloud but with heavy computational and storage costs.

Besides cloud rendering, there are a number of studies of smoke simulations with light scattering implementations. In Figure 4.16(a), Zhou *et al.* [123] use Radial Basis Functions(RBF) to approximate the volume densities of smoke. This uses both image-based lighting and multi-scattering with limited computational cost during the ray casting so as to achieve a real-time performance. However, this method requires substantial precomputation for radiance transfer, which requires the smoke information for the entire animation sequence. For high-fidelity purposes, Figure 4.16(b) shows a Fourier opacity mapping approach advocated by Jansen and Bavoil [58]. Light transmittance is defined by Fourier coefficients. This method is used in the game Batman: Arkham

Asylum, which presents shadowing smokes based on a particle system. For performance reasons, Green [43] in Figure 4.16(c) renders dynamic smokes with volumetric shadows using a 'half-angle' slicing method, which is a simple extension of the volume shading technique proposed by Kniss [57]. Although this is a low-fidelity smoke simulation, it requires only a single 2D shadow buffer and still presents a convincing visual result.

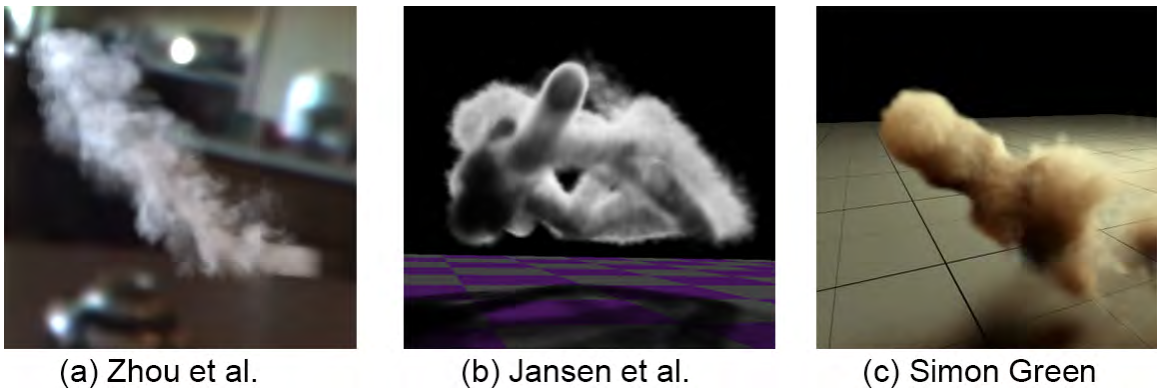


Figure 4.16: Various smoke rendering techniques with volumetric illumination effects.

More recently, there have been many studies focused on complex illumination control for volumetric data. Zhang and Ma [122] present a light propagation method, which support both absorption and scattering via a convection diffusion equation. In this technique, the light simulation and the rendering can be divided into separate stages, which enables flexible control between the rendering quality and its performance.
In general, these smoke rendering techniques present compelling volumetric illumination at real-time performance. Many of these techniques could be extended to cloud rendering as clouds and smoke system use a similar volumetric data structure. However, smoke rendering does have some visual features that are quite different from cloud rendering requirements:

1. Smoke is a highly active medium in which position and shape are constantly changing. Holding a static shape at a specific moment of the smoke animation, which is largely perceived through illumination, is rarely required by the viewers. On the other hand, clouds, although they grow, drift and disperse, generally do so far more slowly. Thus, from a visualisation perspective, rapid change is the key attribute for smoke, while shape holds the same importance for clouds.

2. Simulated smoke tends to be for a single body. However, cloud formations are more complicated. A cloudscape with multiple cumulus clouds at various altitudes leads to complex shadowing effects.
3. Most smoke only needs low-frequency global illumination and low-order scattering. In contrast, one invaluable cloud feature is the capture of high-frequency lighting effects, such as the 'golden rim' seen during the sunset.

These differences suggest that an alternative strategy is required, we propose a fast volumetric lighting and shadow generation method with as little impact on rendering performance as possible. The aim is to provide low-fidelity light scattering on a cloudscape, with the enhancement of shape and overall formation representation.

4.5.2 Our Cloud Illumination Approach

However, a low-fidelity light simulation does not mean a poor shading effect. A cloud with basic scattering illumination can also enhance visual quality, thereby helping the artist to visualize the final shape. Similar to the 'bottom cloud shadowing' painting skill used by the artist in cloudscape painting, as introduced in section 2.2.1, a high contrast shading, such as the 'rim lighting' can largely enhance the shape perception of a cloud. Therefore, our aim is to provide a simple illumination effect that supports the 'rim lighting' effect at minimum computational cost.

Similar to Zhang [122]'s strategy, we divide volumetric illumination into two stages: shadow map generation and illuminance rendering. Once a volumetric cloud is generated, it immediately generates the corresponding shadow map that contains the illuminance value of each cell in the sky volume. Then, the illuminance rendering uses these values to enhance the shading as they also indicate whether the cell is 'rim lit' or in the shadow of the cloud. A comparison is provided in Table 4.1.

Splitting the light shading computation into two parts is based on performance considerations. With the help of Compute Shader, shadow map generation only takes hundreds of milliseconds. This performance is better than most off-line rendering methods, but remains insufficient for a smooth modelling system. Thus, shadow map generation is pre-processed before the rendering. It is only required when texels in the

Table 4.1: Comparison between Shadow Map Generation and Illuminance Rendering.

Shadow Map Generation	Illuminance Rendering
Generating shadow map	Using shadow map for final shading
Time costly	Fast implementation
Occur when dataset changes	Every rendering frame
Running in Compute Shader	Pixel Shader together with ray casting call

volume are altered during cloud mapping or the light source is moved. As with other approaches, we neglect scattering effects because this involves calculating indirect paths.

Shadow Map Generation

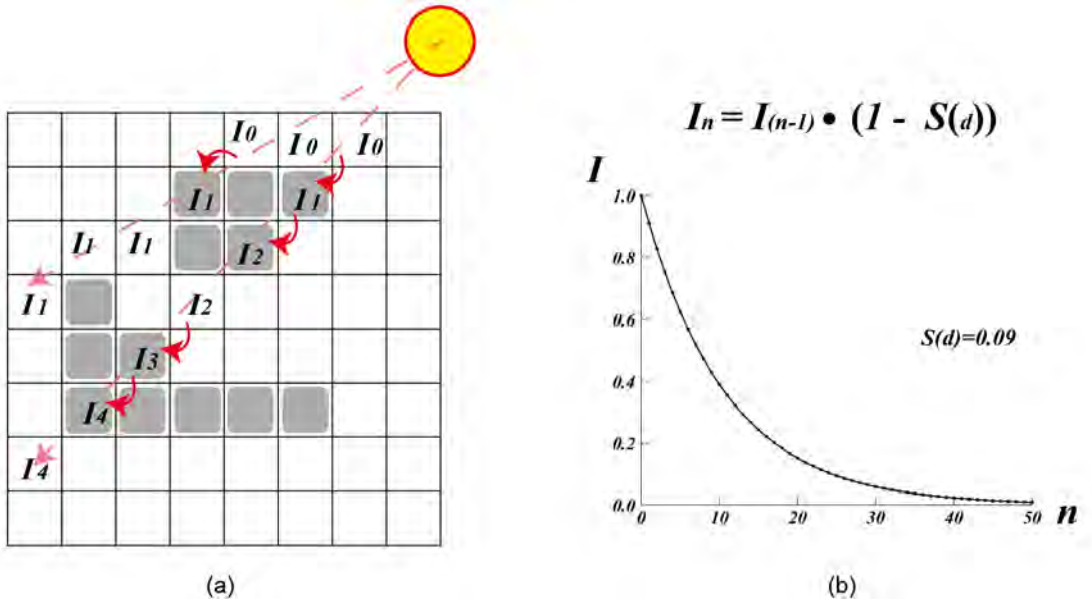


Figure 4.17: Shadow map generation. Each cell contains an illuminance value defined by the light direction and the density of the occupied cells (clouds) along the light path. (a) Any non-empty texel along the path increases the number of attenuation (n) by 1 (b) An example of illuminance variation when the $S(d)$ is a constant value 0.09.

The shadow map generation produce a 3D texture of the same size as the volumetric cloud texture. Each texel of this 3D texture, which represents a cell of the light volume, stores an illuminance value indicating how much light this cell has received from the

light source. Figure 4.17 illustrates the idea of the shadow map generation in a 2D grid of 8×8. In Figure 4.17(a), the illuminance value of each cell is affected by both the light direction and the density of the occupied cells (clouds) along the light path. Any non-empty texel along the path increases the attenuation (n) by 1 and produces the corresponding texel illuminance with the equation:

$$I_n \quad = \quad I_{n-1} \cdot (1 - S(d))$$

$S(d)$ in this equation represents the density value sampled at the current texel, which varies in real cloud modelling. Figure 4.17(b) shows an example of illuminance variation when the $S(d)$ is a constant value of 0.09. This variation indicates that a cell gets completely dark once a light passes through 40 non-empty texels (cloud cells) to reach it. It is also important to note two advantages in using this strategy for shadow map generation:

1. It produces soft shadows as each pixel is sampled by various light paths;
2. The illuminance value can also be used to identify whether texel is in the range of 'rim lighting'. As a texel must be located on the surface of the cloud when its illuminance value approximates to but less than 1.

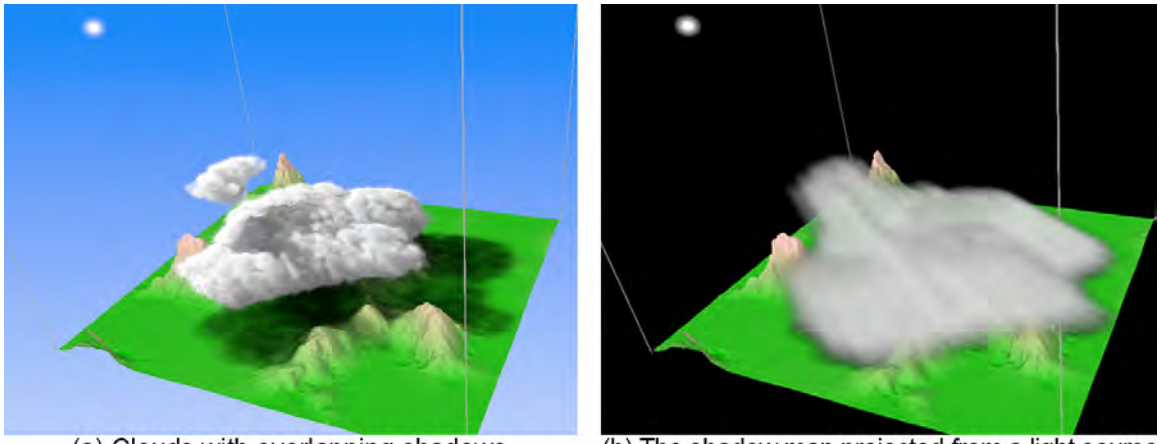
We implement this in a Compute Shader supported by DirectX 11 to leverage the capability of unordered address access to a 3D texture. Considering that the sun is the only direct light source in nature, Algorithm 5 lists the procedure for calculating illuminance values $F_{Illuminance}$ of each voxel inside a light volume. Each $F_{Illuminance}$ depends on the direction $V_{LightDir}$ and the length F_{Len} between the voxel location and the source light position in the light volume. All the non-empty voxels along the light direction absorb an amount of illuminance by $F_{ShadowFactor}$ times the density of the local voxel. Using the Computer Shader, each voxel location can be directly accessed via the DTid address from the 'DispatchedThreadID'. This enhances the efficiency in shadow map generation due to the highly parallel implementations on the stream processors.

Figure 4.18(a) illustrates a cloudscape with cumulus clouds at various altitudes. A light source represented by the white ball projects these clouds from a top-down

Algorithm 5 The shadow map generation on Compute Shader.

```
1: [numthreads(8,8,8)]
2: void CS_SHADOWMAP(uint3 DTid : SV_DispatchThreadID)
3: P_Light ← LightInfo.gba;                    ▷ Get the position of the light source
4: V_LightDir ← P_Light − DTid;                ▷ DTid also refers to the voxel location
5: V_Norm ← normalize(V_LightDir); F_Len ← length(V_LightDir); F_Opacity ← 1.0f;
6: for (i ← 0; i < F_Len; i++) do
7:     P_crtVoxel ← DTid + i · V_Norm;
8:     if OutsideVolumeBox(P_crtVoxel) then
9:         break;
10:    end if
11:    F_Illuminance* = 1.0f − F_ShadowFactor · TexVolumeData.Sample(P_crtVoxel).r;
12: end for
13: txtUAVVolumeData[DTid] ← 1.0f − F_Illuminance;
```

direction and generates the volumetric shadow map, as shown in Figure 4.18(b). It not only produces a soft shadow effect, but also supports shadowing of the top cloud onto the lower clouds. This significantly enhances the perception of shape in complex clouds formations.



(a) Clouds with overlapping shadows (b) The shadow map projected from a light source

Figure 4.18: Clouds on various altitudes with overlapping shadows.

Illuminance Rendering

The volumetric shadow map is processed only when the cloud data is updated. Its associated 3D texture is located in GPU memory, which can be loaded for illuminance rendering immediately. Our illuminance rendering uses a fast cosine transform equation, as shown in Figure 4.19.

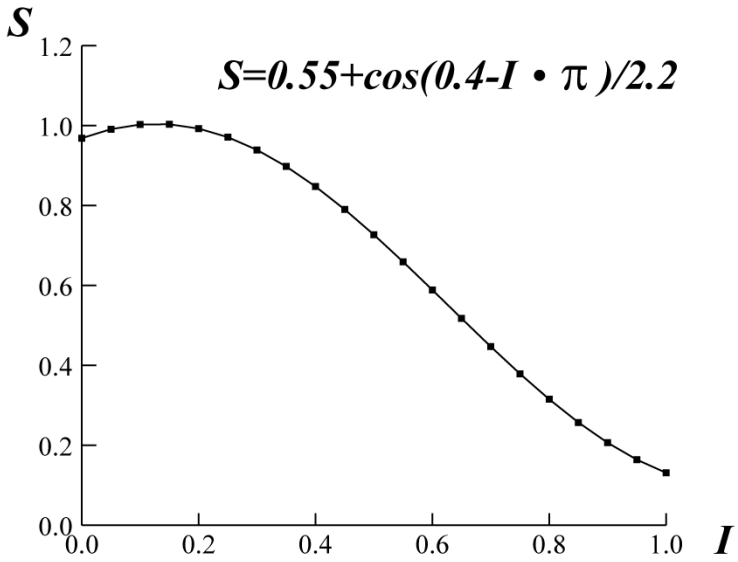


Figure 4.19: The equation of color strength (S) in relates to the illuminance variable (I). Note the (I) value is an inverted value from the shadow map generation, which is indicated in Algorithm 5, line 13.

This equation transforms the illuminance value (I) of each texel from the generated shadow map to a strength value (S) that is used for the final color blending. We designed this equation to yield three benefits:

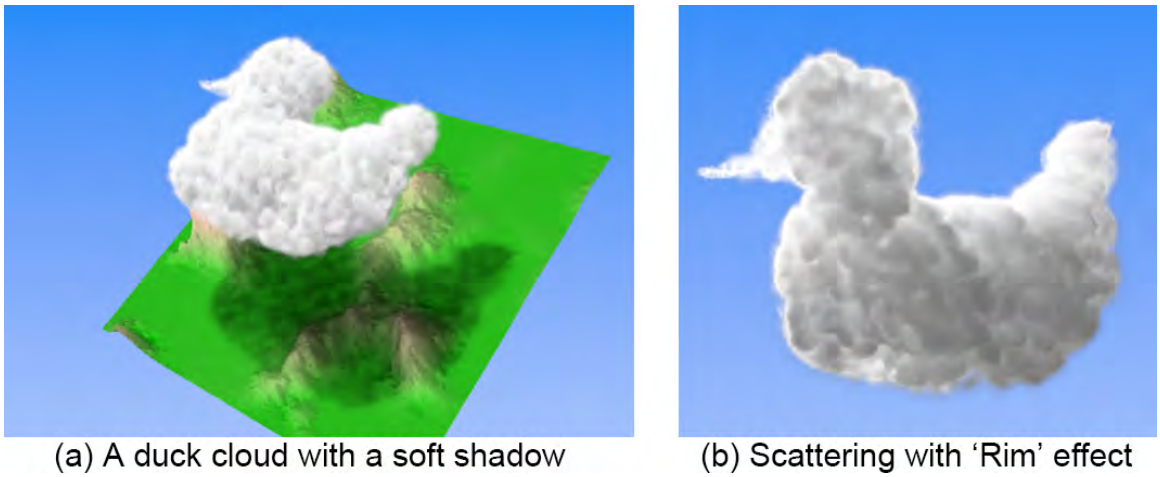
1. It is simple and quick to implement;
2. It provides a smooth attenuation curve, and
3. The curve rises in the beginning to achieve the 'rim lighting' effect, which enhances the shape perception in our cloud modelling system.

The implementation of this illuminance rendering is straightforward; one only needs to override the $ColorBlend()$ function of the ray casting in Algorithm 4, section 4.3.

Once a ray reaches a certain texel, both the cloud density F_d and illuminance value I are sampled from two separate 3D textures: cloud data and shadow map textures. We merge the colour of the previous ray step, C_{pre}, with this sampled data (F_d and I) to derive a new colour, C_{new}, for the current step by fast cosine, as follows:

$$\begin{aligned} S &= 0.55 + cos(0.4 - I \cdot \pi)/2.2 \\ C_{new}.rgb &= S \cdot F_d + (1 - F_d) \cdot C_{pre}.rgb \\ C_{new}.a &+= F_d \end{aligned}$$

The results of this approach are evident in the rim lighting of the duck-shaped cloud in Figure 4.20(c).



(a) A duck cloud with a soft shadow (b) Scattering with 'Rim' effect

Figure 4.20: A duck shape cloud using our illuminance rendering.

4.6 Summary

In this chapter, we further explained our method for volumetric cloud rendering. This rendering method provides convincing cloud visualisation with the support of volumetric soft shadows and enhanced illuminance features, particularly suited to clouds.

During the development of the volumetric renderer, one major problem encountered was low frame rates due to the heavy computational cost of ray casting. To tackle this problem, we proposed a GPU-based octree generation and traversal method to detect the empty space inside the volumetric data. As a consequence, our cloud rendering method runs at interactive rates. This provides a strong performance foundation for the simulation of a Chinese brush, which is the subject of the next chapter.

Chapter 5

Brush Simulation

Realistic brush simulation is difficult due to limited support from input devices, as the simulation of bristle deformations (split, bend, twist) remains challenging for any existing digital device. Instead of a high fidelity simulation, we aim to achieve similar footprint features using straightforward approaches. This chapter presents three approaches to Chinese brush simulation: a skeleton-based brush, a 2D brush and a 3D brush. They are implemented using various techniques and evaluated by both Chinese calligraphers and digital artists.

In the background chapter, we reviewed several brush simulation techniques; we also presented some ancient Chinese calligraphy, as well as several traditional cloudscape paintings. This leads to the observation that a number of brush features can be abstracted from their footprints such as diversity, simplicity, intricacy and elegance. In general, paintings are the creative products from the use of these brush features, while calligraphy reflects the level of mastery a calligrapher achieves. In order to better understand the process by which the Chinese brush produces these effects, and most importantly, the implications for cloud modelling, we further study the simulation of such a brush using a pressure sensitive pen device.

5.1 Basic Strokes for Calligraphy

We begin by introducing eight basic strokes, which are listed in the Figure 5.1. Most Chinese characters are formed by the combination of these strokes. These eight strokes are in a font called 'Kai', which is a regular script for Chinese handwriting. It is one of the most commonly used fonts in Chinese society.

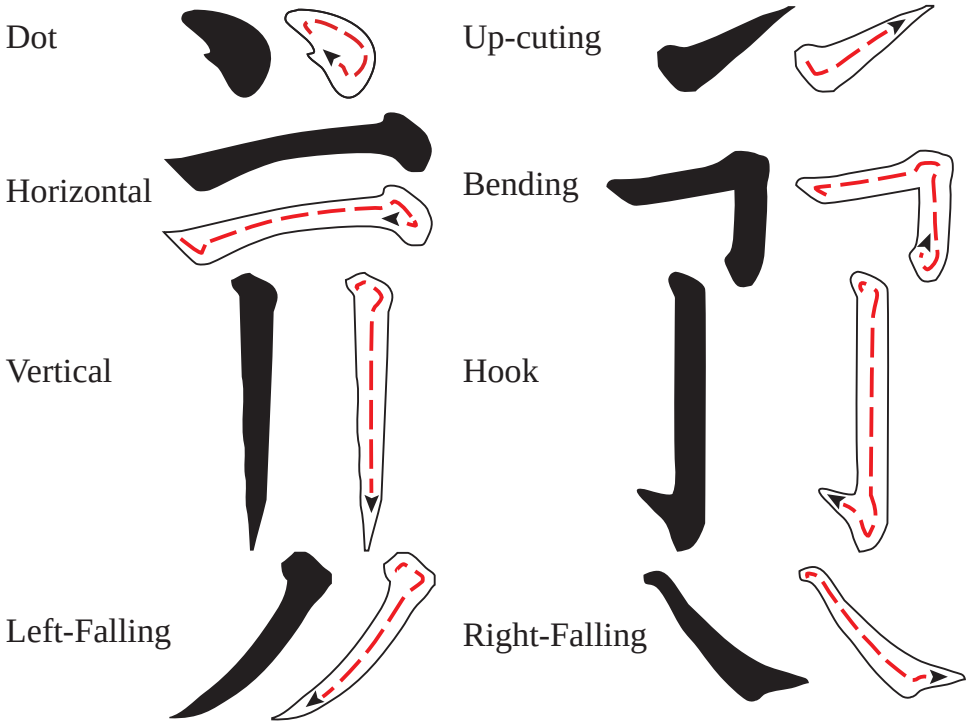


Figure 5.1: Eight basic strokes for Chinese characters, written in a regular handwriting style. The arrowhead of each stroke indicates the direction of the drawing with its path presented by a dotted line. Note the intricacy of drawing a stroke, which produces an aesthetically pleasing footprint with a harmonious start, smooth body and natural ending.

Figure 5.1 relates each of the stroke's footprints to its writing direction as a dotted red line. Among these footprints, the first stroke is a simple dot; five of the other ones are straight lines but point in different directions. Only two strokes, bending and hook, involve two directions in one stroke. Normally, mimicking these strokes using a pen takes little time. However, producing them correctly using a Chinese brush requires several days of training.

The directions in Figure 5.1 indicate that brush flow is more intricate than its footprint

suggests. The initial direction of the brush always starts opposite to the prevailing direction of the stroke. Then, the brush makes an immediate turn and follows the stroke direction. In the end, the brush turns back to the opposite direction to finish the stroke. This reversal happens in the stroke 'Dot', 'Horizontal' and 'Bending'. This intricate drawing style begins the stroke footprint with a harmonious start, continues with a smooth body and ends the flow naturally. It follows a similar philosophical idea to Taichi practice, where a powerful punch strike can be resolved through a small supplementary force that is applied perpendicularly to the attacking direction. A stroke with a simple but boring line can be embellished through a small bristle press that is applied orthogonally to the line direction in the beginning of a calligraphic stroke.

5.1.1 Stroke Intricacy and Variation

Writing using a Chinese brush is different from using a pen. Firstly, the way of holding a Chinese brush is unique, as shown in Figure 5.2.

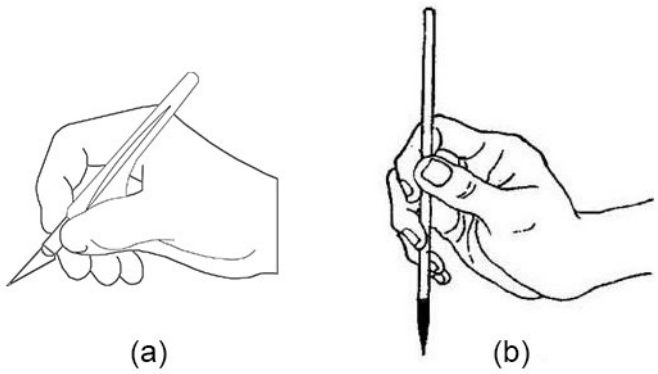


Figure 5.2: Pose for holding a pen (a) and a Chinese brush (b). Note the wrist is suspended over the table when holding a Chinese brush.

Normally, a pen is held using a tripod grip. Three fingers form a tripod to support and control the movement of the pen. At the same time, the wrist is resting on the table. Figure 5.2(b) shows that a Chinese brush is held using a *clipping grip*. Four fingers are involved in supporting the brush. The forefinger and the middle finger are located on one side, while the thumb and the ring finger are on the other side. The

brush is held vertically while the wrist is suspended over the table. In practise, using a brush is more tiring than using a pen. However, a suspended wrist freed from the table plus a vertical grip enables a more complex drawing stroke.

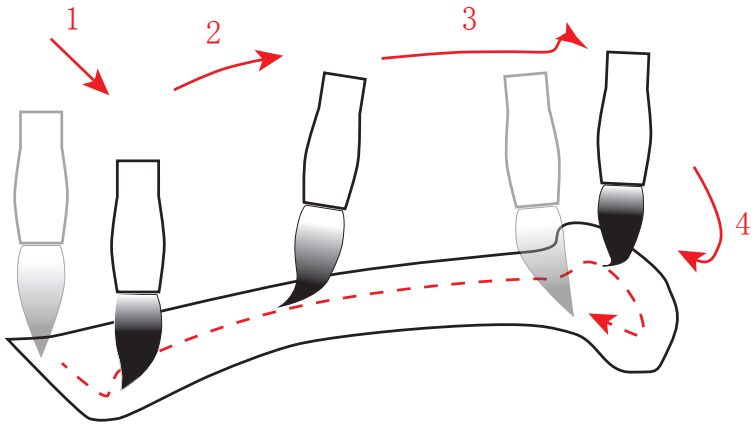


Figure 5.3: A detail brush sequence for drawing a horizontal line. Note the precise control of pressure and flow.

Secondly, the way a stroke is drawn using a Chinese brush is also different from using a pen. Figure 5.3 uses a horizontal stroke example to demonstrate this complexity. Unlike drawing a straight line using a pen, drawing this horizontal stroke using a Chinese brush is separated into four steps:

1. Press and slant the bristles in a left down direction.
2. Drag the brush along a horizontal flow.
3. Move the brush up and right and immediately twist it downwards.
4. Turn the brush from downwards to the left and lift the brush up to end the stroke.

Among these steps, both the beginning and ending portions deviate substantially from the main stroke flow. In calligraphy, this intricate flow mechanism is widely used to enhance the aesthetics. From a philosophical perspective, a single line forms a shape, a shape contains a pivot, a pivot generates balance, the balance leads to the beauty. Thus, all the four steps in this stroke produce various shapes at the four corners. After step one, the tip of the bristle forms the top left of the stroke (a sharp corner) while

the body of the bristle enables the bottom left of the stroke (a round and smooth shape). The second step generates the main section of the horizontal stroke. The third and the forth steps conclude the stroke using a circular flow.
In this way, a well-balance stroke is generated with four of its corners in various forms. This font style is considered one of the formal writing styles that provides the most beauty in calligraphy. All in all, this intricate horizontal shape is a single stroke that requires less than a second or two to draw.

5.1.2 Features Summary

Similar to the correct use of chop sticks, the correct use of the Chinese brush is not as easy as the resulting footprint suggests. We describe five major mechanisms needed for the control of the Chinese brush:

1. **Pressure control** that flattens the bristles and thickens the strokes;
2. **Flow control** that twists the bristles and splits the bristle tip;
3. **Speed management** which affects the ink dispersion and so determines stroke appearance;
4. **Tilt of the brush** changes the surface of contact, which significantly influences the shape of a footprint;
5. **Ink dispersion effect**, which leads to a blurred edge.

We aim to simulate the Chinese brush through the control of these five mechanisms. In our modelling system, we use a Wacom's Intous 4 tablet as the input device. Its stylus supports 1024 levels of pressure control and a tilt angle of close to 90 degrees. Although this state-of-the-art device is widely used in the animation industry and graphics design, it remains very difficult to produce a calligraphic footprint using such a device.

5.2 Initial Brush Simulation using Skeletons

Similar to other skeleton-based brush implementations [5] [13], which utilise a natural spring effect from the bristle bending, our initial design uses a skeleton-based method

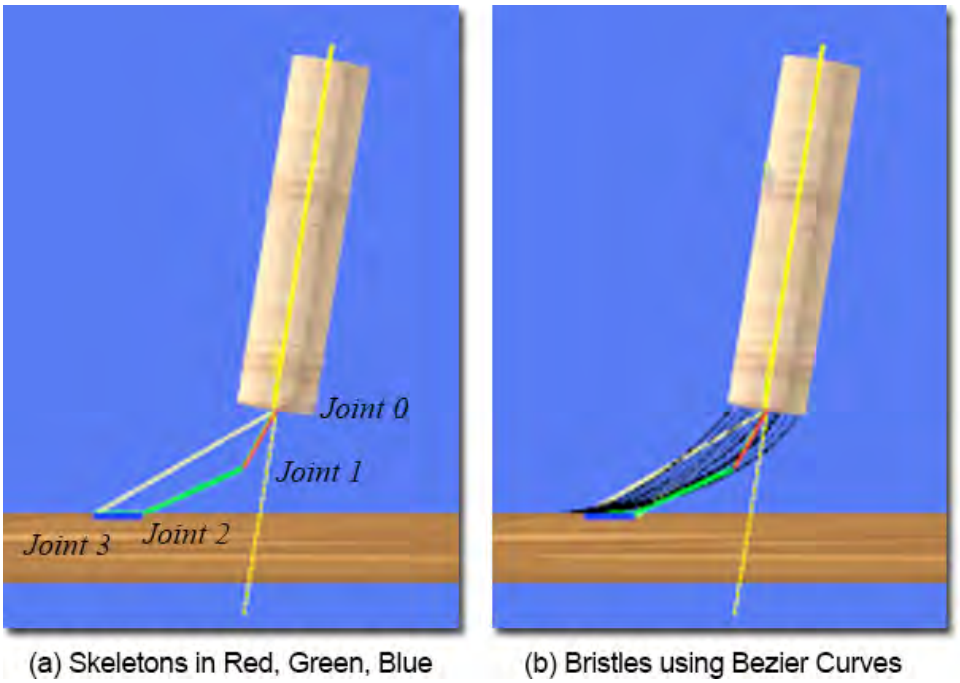


Figure 5.4: The skeleton brush contains a main spine with 3 segments. 4 joints are used for bristle generation on Geometry Shader.

to simulate the Chinese brush. The skeleton consists of 4 joints and three line segments as shown in Figure 5.4(a). Once the bristle touches the table, the location of these joints are calculated using a Point Verlet integration [4, 87].

5.2.1 Joints Control

The Position Verlet we implemented describes the movement of a point through time. One of the main advantages is that it is unnecessary to calculate the velocity of the points during the movement. At each time frame, it requires four variables: the position of the current frame (P_{crt}), the position of the previous frame (P_{prv}), the predefined acceleration vector (V_{acc}) and the time step between the current and previous frames (T_{step}). These variables are processed using the following equation:

$$\begin{aligned} P_{Expected} &= 2 \cdot P_{crt} + (P_{crt} - P_{prv}) + V_{acc} \cdot T_{step}^2 \\ P_{prv} &= P_{Expected} \end{aligned}$$

However, Point Verlet works by resolving the $P_{Expected}$ value closer and closer to the correct position through a number of iterations, with 5–7 iterations usually providing a rigid result. For the locating of four joints, we implement 10 iterations on this equation at each frame to satisfy the $P_{Expected}$ value subject to a constraint function that keeps the distance between each of the 4 joints fixed during the movement.
The position of the rod, *Joint 0* in Figure 5.4(a), follows the movement of the stylus pen in x and z coordinates, while its y value is defined by the pressure and the tilt level of the pen. The other 3 joints are located via iterative solution of the Position Verlet. Then, the updated joints locations are passed to the Vertex Shader as the input vertices. They are then grown into bristle vertices in a Geometry Shader as the control points of Bezier curves, similar to the method used by Lu *et al.* [74]. Thus, the splitting of the bristle can be achieved through the tilt of the skeleton, which is controlled by the pen pressure.

5.2.2 Footprint Rendering

Ink dispersion is one important feature in brush calligraphy. A number of studies [13, 120, 69, 65] aim for high fidelity simulation of this effect. However, in our cloud modelling system, the 2D footprint from the brush simulation is not at the end of our workflow. This footprint is actually the entry point for cloud generation, in which the cloud data is dispersed in a 3D volume using a different strategy. It is therefore unnecessary to simulate a high-fidelity ink effect at this stage. Any new coloured texels from the dispersion on the 2D footprint may lead to extra shape growth during the later cloud generation stage. Considering the five features we summarized in section 5.1.2, the 2D footprint is required to preserve the raw information of the bristle movements, thickness and pressure variation. We implement footprint rendering in three steps:

1. A top-down camera is set to implement an orthogonal projection (Viewing at the paper from top);
2. The depth of the view frustum is clipped so as to capture the Bezier curves only above the surface of the table;

3. The frame buffer is left uncleared from the previous rendering;

This frame buffer, as the footprint, is then stored to a 2D texture which captures the movement of the Bezier curves across all the rendered frames.

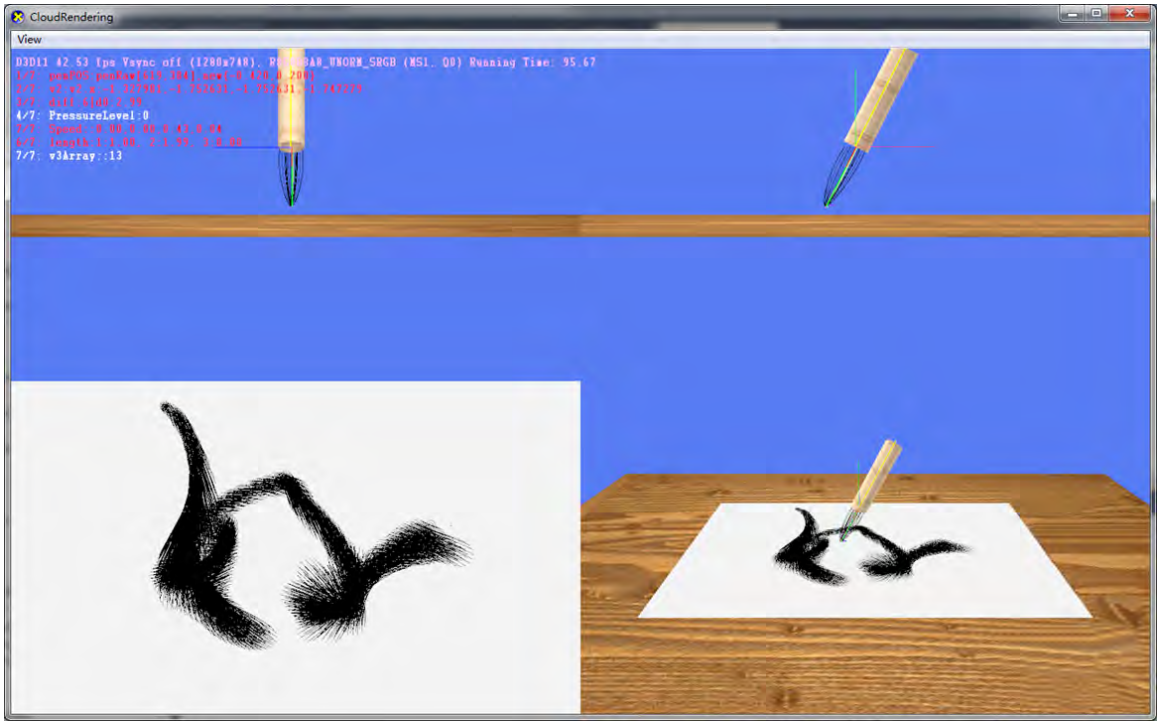


Figure 5.5: The interface for skeleton-based brush drawing. Both the brush status and the footprint are tracked.

Figure 5.5 shows screenshots of our skeleton-based brush simulation. The bottom viewpoints show the 2D footprint from both the top and perspective views. In addition to the thickness variation, it captures a unique hairy feature located at each turning of the stroke, which can help to define a cloud with a wispy surface.

In order to test this skeleton-based brush, we recruited four volunteers to participate in our observation-based evaluation. The first two volunteers were postgraduate students from Human Computer Interface research field. And the next two were experienced calligraphers who have continuously practised both calligraphy and Chinese painting for more than 40 years. These volunteers were evaluated individually. They were required to draw three Chinese characters using this skeleton-based brush and share

their opinions. After four evaluations, we made a decision to stop the test and recognized the failure of our skeleton brush design. As all the users found it confusing to draw characters using this brush style right from the beginning and none of them accomplished all the requested three characters. The main problem is that users found the motion of the bristle tip difficult to predict. Normally, when using a stylus pen, the footprint starts from where the pen tip touches and follows the tip flow. Unfortunately in our skeleton-based brush simulation, the pen tip is not correlated to the bristle tip. It corresponds to the base of the bristles, the bottom of the brush rod. No matter how realistically we simulate the movement and the deformation of the bristle by using pressure and tilting values from the pen, the location of the bristle's tip is always different from the top of the bristle, where the pressure pen touches the tablet. This is disconcerting to the users.

5.3 Texture-based Painting

To solve the stylus tracking problem, we adopt another stratagem for brush simulation. Instead of simulating the bristle motions, the footprint can be generated directly from a texture-based brush style, an approach which is widely used in existing painting packages. In these applications, the center of the footprint sampling snaps to the location of the stylus pen. Users can paint at a precise location through the use of a stylus pen. We implement two texture-based painting styles to emulate the brush features: 2D and 3D brushes.

5.3.1 Implementation of Texture-based Brushes

Figure 5.6 illustrates the workflow of each frame in the *Painting Mode*. The left part lists all the stages that take place in this frame, while the right part shows the subprocess of the footprint generation stage. In each frame of the painting mode, footprint generation and its rendering work on the same 2D texture but for various objects. The footprint generation aims to update a 2D texture based on the uncleared footprint data left from the previous frame. In contrast, the footprint require the

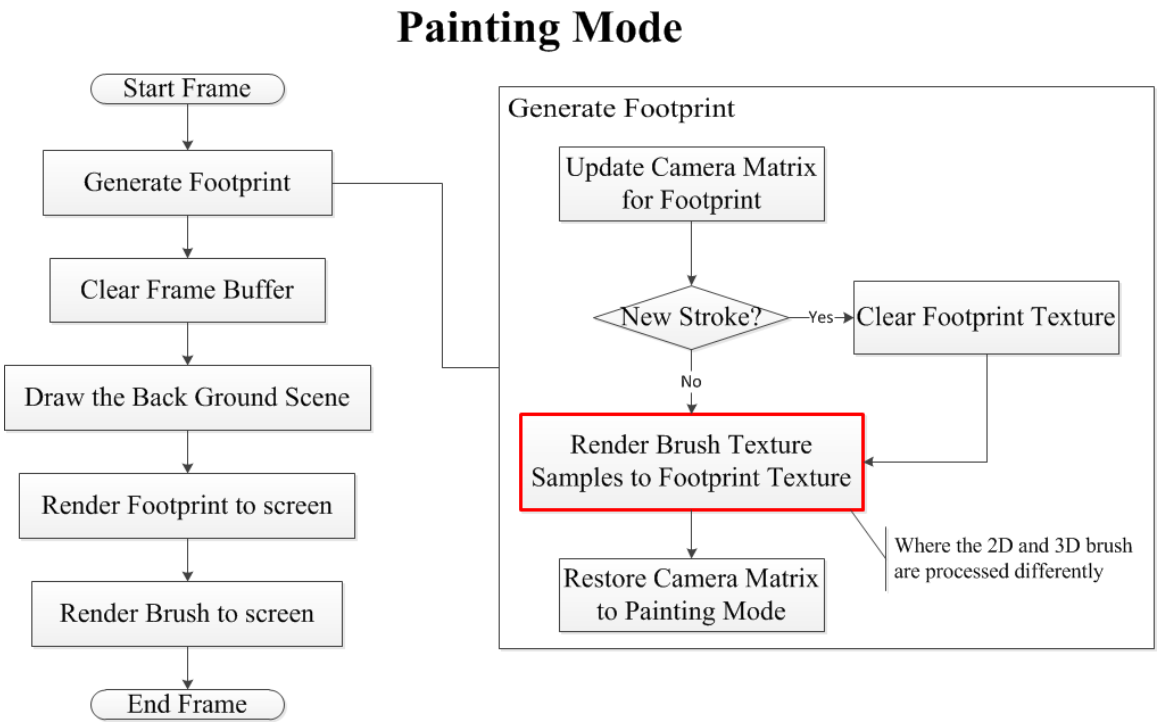


Figure 5.6: The workflow for the painting mode. For each rendered frame the functions on the left are executed sequentially. Both the 2D and 3D brush styles are executed in the same procedure. The only variation occurs in the highlighted step, where different footprint samples are used for the two brush styles.

screen buffer to be cleared so that an updated footprint texture can be rendered to the screen together with the background scene and the brush object. Both our 2D and 3D brushes follow the same workflow. The only exception is when rendering brush samples to the footprint texture, which is outlined in Figure 5.6 with a red box.

The 2D brush uses a static 2D texture sample (shown in Figure 5.8(a)) in a similar fashion to the brush mode in Photoshop, in that texture samples are splatted into screen space repeatedly as the stylus pen moves. The size of the sample changes according to the pressure of the pen. Similar to footprint capture in the skeleton brush, the frame buffer is left uncleared so as to record all the rendered frames. The main difference is that it records the Bezier curves for the skeleton brush, but captures images for the texture-based brush. The other variation is that in the 2D brush, the tilt of stylus pen has no impact on the texture sample.

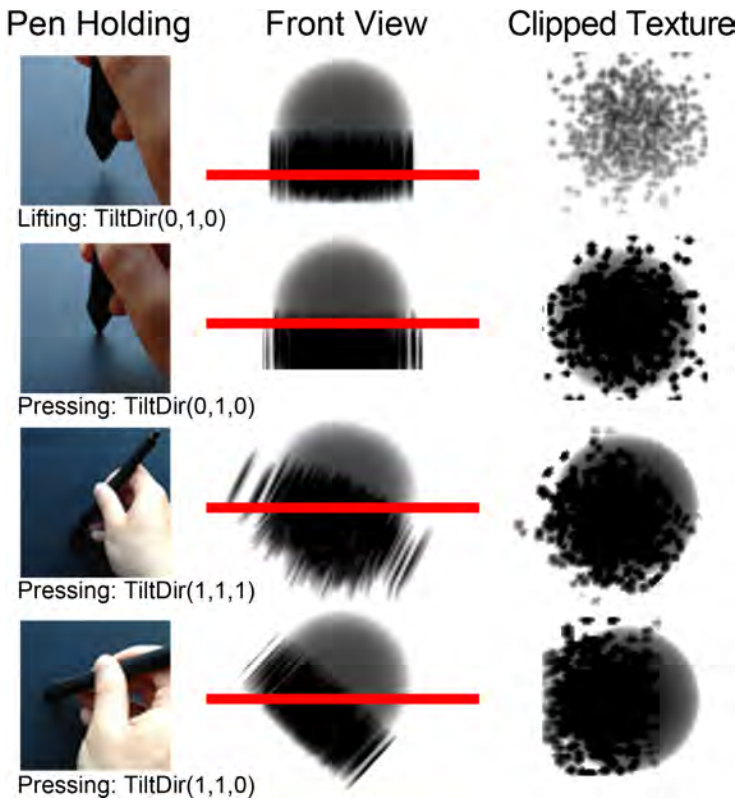


Figure 5.7: The dynamic 3D brush, showing how stylus pen tilt and pressure control a cutting plane through the 3D texture that is used to create a clipped 2D footprint.

The 3D brush operates by sampling slices from a volumetric texture (see Figure 5.7). The normal direction of the slice is determined by pen tilt and the height of the slice is determined by pen pressure. These two parameters – vertical position and normal direction – determine a cutting plane against which the 3D texture is then clipped in real-time to a 2D texture, followed by splatting to screen as before.

From the front view column in Figure 5.7, the volumetric texture holds a 64^3 3D ink dataset. This dataset has two parts, a solid sphere that fades at the edges and a noise sample that extends along the y coordinates. This enables various footprint samples produced from the different pressure and the tilt control. The footprint is sparse and noisy when the pressure is soft. It can also be more solid when the pressure is hard. If the pen is pressed an any angle, the tip of the pen generates a noise sample. At the same time, the other side of the intersection generates a rounded footprint as

the clipping plane cuts through the spherical part of the ink dataset (bottom row in Figure 5.7). In this way, the 'tip splitting' effect, which occurs when the brush is pressed down hard can be emulated through the control of a stylus pen.
The footprint from the clipped section also uses the volumetric rendering technique we discussed in Chapter 4. However, a plain volumetric rendering is implemented for this 3D ink dataset with the exception of the octree and light illumination functions. Furthermore, the plain volumetric rendering only marching through a thin slab of the ink volume depending on the clipping plane, which is usually 10 times smaller than the 64^3 ink dataset. As a consequence, the performance of painting in the 3D brush style is stabled at 50-60 fps with 80-90 fps for 2D brush style.

5.3.2 Footprint Comparison

Figure 5.8 shows a comparison of footprints generated by the 2D and 3D brushes. These footprints are for the same Chinese 'dragon' character. They have some common features such as varying thickness along the stroke and a dispersed ink effect related to brush velocity. Apart from these features, the major difference between the two styles is in the edge patterns. The 2D brush footprint in Figure 5.8(a) has a striated effect aligned with stroke direction and is reminiscent of cirrus clouds. In contrast, the 3D brush footprint in Figure 5.8(b) forms a solid stroke body with a wispiness radiating outwards from the edges that most closely matches cumuliform clouds.
Using these two painting styles, the center of the footprint texture corresponds to the tip of the pen, which accords better with user intuition. With the exception of the ink dispersion effect, the other four features of Chinese brush calligraphy can be replicated by the users. Unlike traditional calligraphy training, these painting styles are easy to use and can be mastered in a couple of minutes (as evidenced by our user studies).

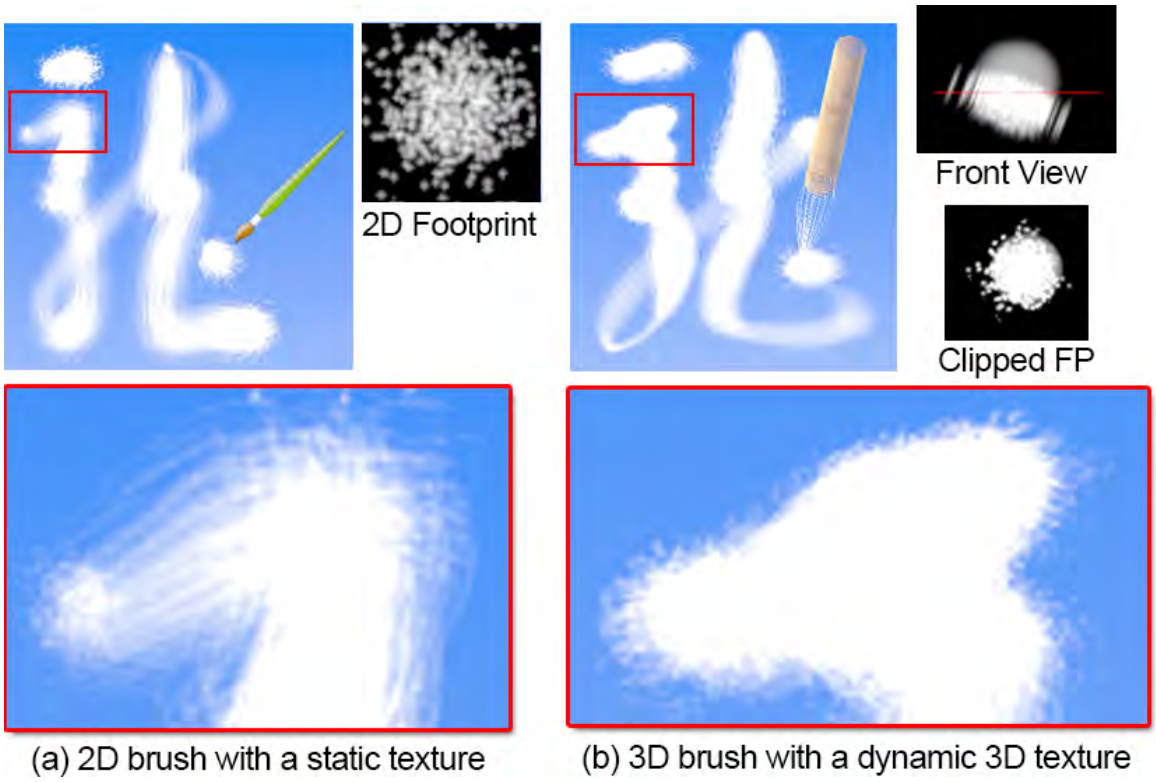


Figure 5.8: A footprint comparison using (a) 2D and (b) 3D brush styles.

5.4 Summary

This chapter began with an investigation of Chinese brush simulation using a skeleton consisting of 4 joints, surrounded by bristles represented as Bezier curves. Unfortunately, this proved confusing for users, sinces counter to their expectations, the brush tip did not track the stylus tip. Instead, we developed two texture-based painting modes: a 2D brush, with a static 2D sampled texture; and a 3D brush, with a dynamic 2D texture derived by taking a cross section through a 3D texture.

The next step is to convert the cloud-like 2D footprint into a volumetric cloud with a density and appearance that matches the input stroke. This needs to be done both automatically and efficiently.

Chapter 6

Cloud Growth

An important aspect of our new modelling system is the mapping from a 2D brush footprint to 3D cloud data. In this chapter, we develop a new method that allows the modeller to focus on traditional 2D painting input while the 3D mapping occurs automatically.
We start with a ray-marching mapping method, which enables real-time cloud generation. However, this method may produce unnatural cloud shapes and we thus propose a *Hybrid Mapping Method (HMM)*, which generates 3D clouds immediately after each stroke is painted. This approach produces compelling cloud shapes with various surface features for different cloud types.

6.1 Initial Mapping Method

In order to easily and efficiently generate 3D clouds from 2D footprint input, we seek to achieve 3 goals:

1. A fast mapping process which maintains interactive update rates during modelling.
2. The mapping results should accord with the 2D footprint and therefore reflect the designers' ideas correctly.
3. The procedure of mapping should be easy to understand so that a designer can use it in an intuitive way.

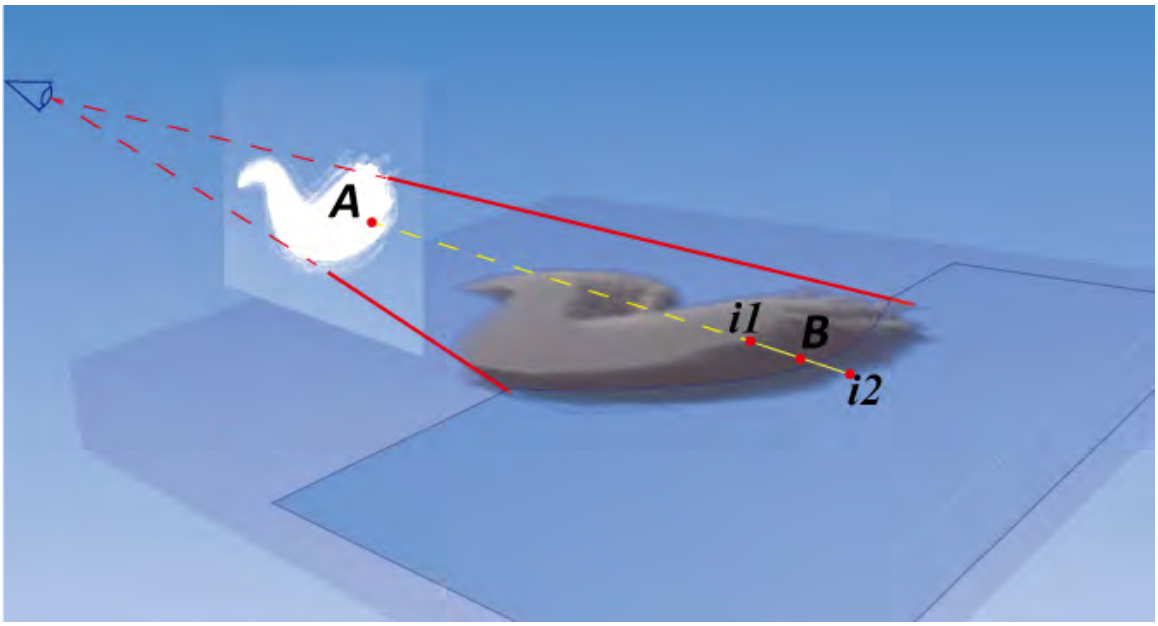


Figure 6.1: The initial mapping method in our system, which uses a ray marching strategy. The length of the mapping ray in the 3D canvas is defined by the density value A in the footprint. The generated volumetric cloud has an unrealistic shape.

Figure 6.1 shows the strategy used in our initial design. In this approach, the mapping is implemented using a ray marching method. Once a 2D footprint is generated by the artist, all the non-empty pixels of the footprint initialize a ray into the 3D data using the current view and projection matrices. These rays then intersect with a 3D canvas, which is predefined by the artist in Canvas Mode. Starting from the intersection point (Point B in Figure 6.1), data voxels grow outwards along the yellow ray for a length corresponding to the alpha value of the pixel (Point A). In this example, the alpha value of A is one, which maximizes the length between the ceiling and the floor of the 3D canvas. Point *i1* and Point *i2* are the intersection points of both the ceiling and the floor. In this way, a volumetric cloud is generated based on a 2D footprint.
There are three advantages to this method:

1. The color density of a pixel of the footprint defines the growth of voxels along the ray. This mechanism, similar to pressure control when using a real brush, enables an intuitive modelling method for the artists.
2. The implementation of this method merges well with the volumetric rendering

mode in our system, which leverages the power of the GPU and provides acceptable performance.

3. It enhances the fidelity of a cloud, especially for complex edge effects, since it captures all the non-zero values in the footprint and interprets them as the thickness of the cloud.

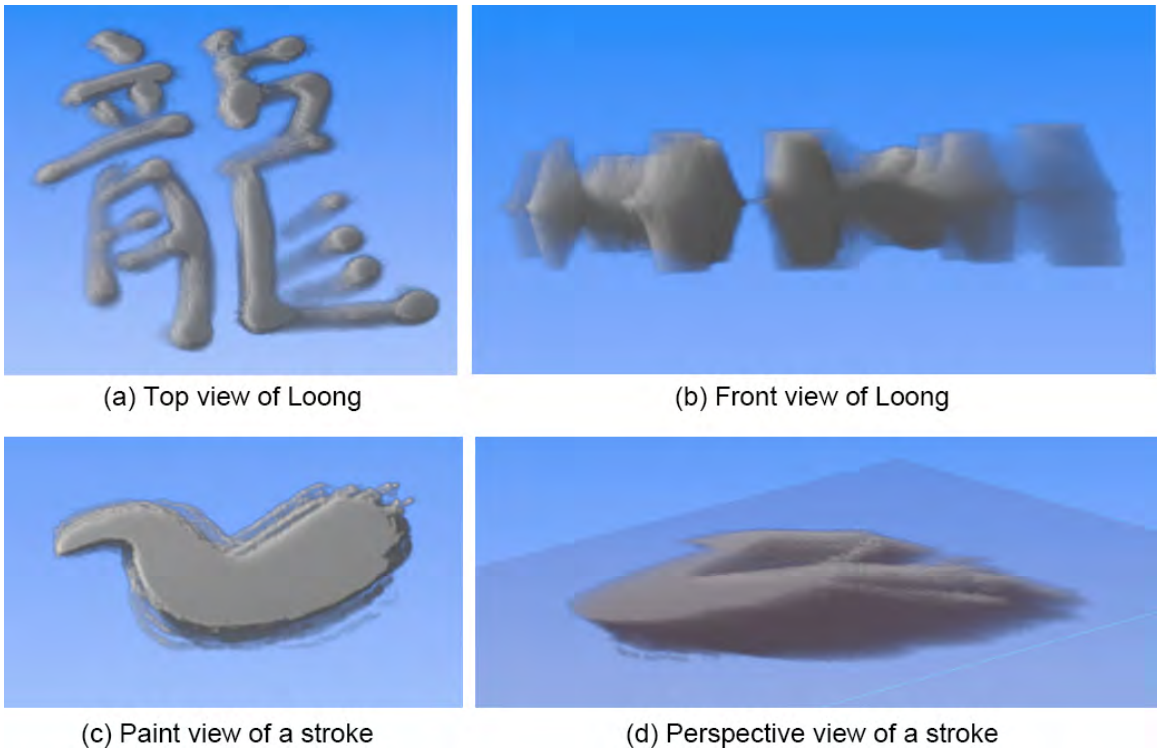


Figure 6.2: Two sets of examples show the shortcomings in the initial mapping method: a vertical symmetry problem for the Chinese character 'Loong' and a consistent distortion in a stroke.

However, shortcomings are also obvious when using this method. Figure 6.2 illustrates two problems in using the initial mapping method. The top two images present mapped volumetric data of the Chinese 'dragon' character. Figure 6.2(a) shows the character from the top view direction, which matches closely with its 2D footprint. However, when looking at this 3D data from the front view, as shown in Figure 6.2(b), an obvious vertical symmetry is evident, which is unnatural for real clouds. The two lower pictures show the same stroke as in Figure 6.1. We include Figure 6.1 here to illustrate a major difference between the mappings of these two groups: 'dragon' is

mapped from a top view direction, while the lower stroke is mapped from the paint view to the 3D canvas at a fairly slanted angle, as shown in Figure 6.1. This slant also leads to a relatively unnatural looking cloud, as can be seen in Figure 6.2(d). It shows a consistent slope across the entire cloud.

6.2 Overlapped Metaballs

Table 2.2 in the background chapter illustrates a number of methods for cloud shape generation. Circles or metaballs are widely used in modelling and rendering systems for objects like smoke and clouds. Nishita *et al.* [84] initiates the cloud modelling using overlapped metaballs. Dobashi *et al.* [26] and Roditakis [99] successfully implement metaballs in cloud datasets built from satellite images. Metaballs refer to the density of the cloud region that the satellite captures. Each metaball is rendered as a semi-transparent circle with a feathering edge and overlapped in a formation which covers thousands of square kilometers. From the perspective of meteorology research, scientists are more interested in the overall formation of the clouds, their altitude, coverage and movement, etc. The visualisation of an individual surface is of secondary importance.

In addition to the use of real cloud data, Ebert *et al.* [32] propose the exploitation of metaballs to form the coarse shape of the cloud. Many systems use metaballs as proxy geometry to define the shape of a particle system. However, managing the locations of these metaballs is often tedious for artists, and the generation usually runs off-line.

Figure 6.3 demonstrates the evolution of a smoke simulation system. In Figure 6.3(a), three solid and separated metaballs provide a poor representation of smoke. Once a noise function is implemented and sampled in these three balls, as shown in Figure 6.3(b), a better impression of smoke is achieved. With more metaballs overlapped in Figure 6.3(c), the smoke effect is further enhanced.

Once these overlapped balls start moving and expanding, the rendered result is highly convincing. However, we encountered some issues when implementing these ideas in our system.

1. Using metaballs only to form a shape or formation in a 3D cloud is not as

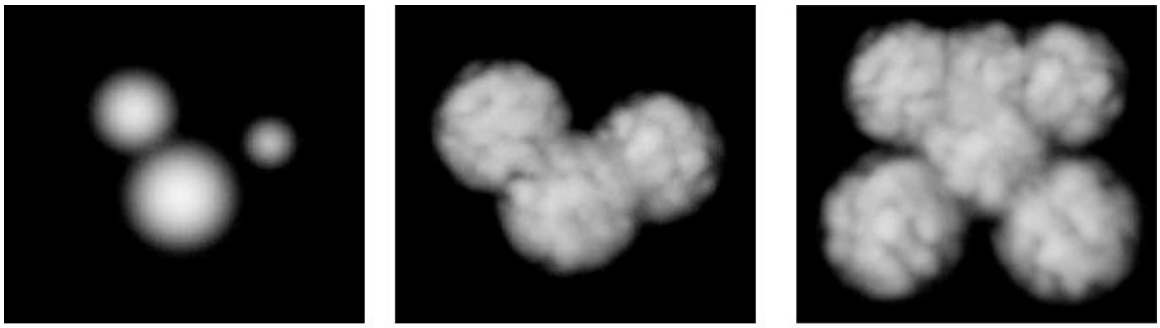


Figure 6.3: Overlapped circles as proxy geometry for smoke generation and rendering. (a) Three smooth metaballs provide a poor representation. (b) Overlapped metaballs with noise surfaces. (c) Smoke effect is enhanced with more metaballs.

convincing as with smoke, since clouds are relatively static. It is easier to spot the regular outline of circles if a shape is stationary.

2. The metaballs in these smoke simulations are procedurally generated in relation to the time and direction of emission. Users have limited control over their location.

We thus propose a new method to map a 2D image into a 3D dataset, the *Hybrid Mapping Method (HMM)*. This mapping uses both overlapped metaballs and the original 2D image. It generates a comprehensive 3D dataset that captures all the details from the 2D source image. Furthermore, it provides interactive performance, which is important since our system aims to support good interaction with the artists. The mapping occurs once the artist finishes drawing a stroke. This provides immediate feedback for the artist, thereby enhancing the user experience. Thus, the performance of the mapping generation is vital in our system.

6.3 Hybrid Mapping

Our Hybrid Mapping Method (HMM) consists of both body and edge generation from a given 2D footprint. Our goal is to allow artists to focus on their brush work and leave the 3D cloud generation to the system, which should not burden the user with

parameter setting. The mapping process should also be able to generate appropriate voxel data at interactive rates.

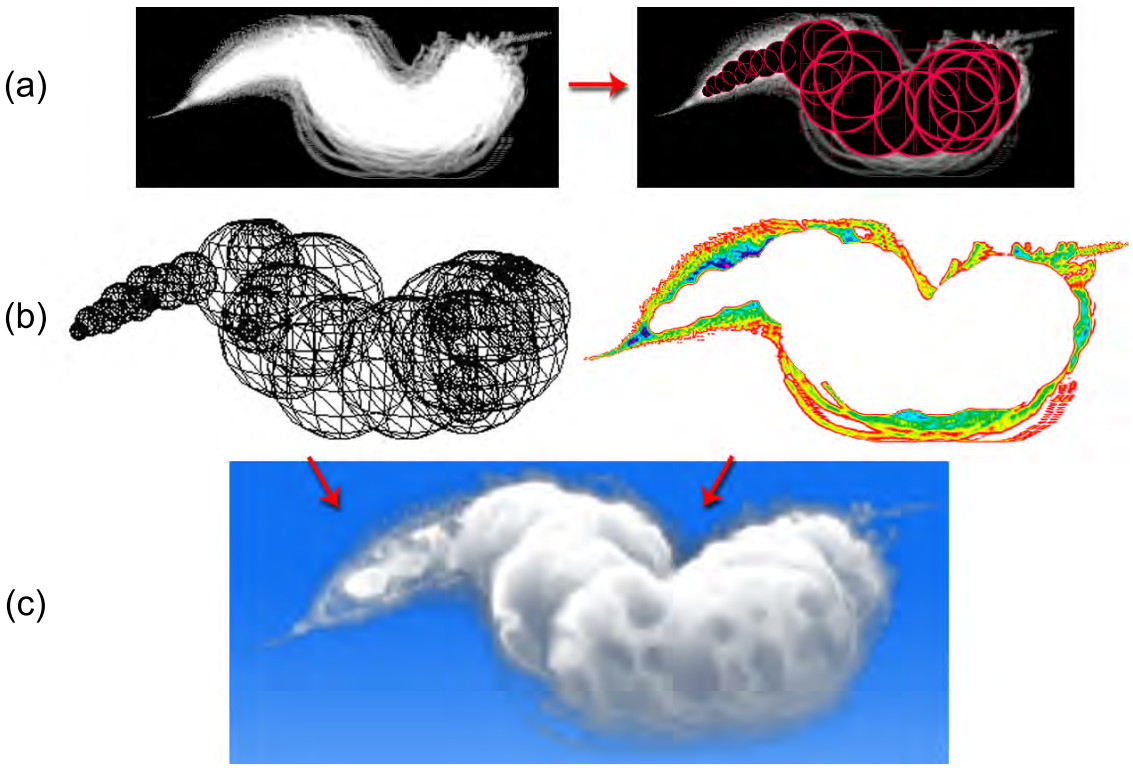


Figure 6.4: Stages of Hybrid Mapping: (a) circle growth; (b) sphere and edge mapping; (c) sphere and edge merging.

Figure 6.4 shows the structure of our HMM, which is divided into three stages:

(a) In the first stage, a collection of overlapped 2D circles is generated based on a given footprint. The radius of these 2D circles is based on the proportion of white (cloud) pixels relative to the area of the circle.

(b) Next, circles and the remaining edge pixels are mapped into the 3D scene using two different methods. Spheres corresponding to the circles are intersected with the altitude layer, while the non-empty texels along the cloud edge are mapped to intersect with the 3D sphere surface.

(c) Finally, the sphere and edge texels are merged in 3D voxel space. This involves

sampling a volumetric turbulence noise dataset and generating a volumetric shadow map, to allow final volumetric rendering.

We discuss these stages in further detail in the following subsections.

6.3.1 Aims for the Circle Growth Phase

Section 6.2 mentioned two shortcomings of using overlapped spheres as proxy geometries to represent static object such as clouds. The first issue (an unrealistic regular outline) can be tackled by, for example, randomizing the sizes of the overlapping spheres. However, the main problem, the lack of intuitive shape control, is difficult to untangle given that the formation of the spheres is generated procedurally or using a tedious Constructive Solid Geometry method.

In our *Hybrid Projoction Method*, the shape control mechanism allows for more user control than with procedural generation. A 2D footprint defines the cloud shape and that shape encapsulates a number of overlapped circles with varying radii. We aim to populate the shape using a circle growth approach, with the following attributes:

1. It is able to fill a variety of stroke shapes, such as strokes with varying thickness or a large patch formed by overlapped strokes, etc.
2. The number of generated circles ought to be minimized so as to enhance the performance of subsequent volumetric sampling.
3. The area masked by the grown circles ought to be maximized in order to reduce the number of remaining non-empty pixels in the footprint. This enables faster processing during the later edge mapping stage.
4. The circles should be visually overlapped in an irregular fashion to ensure a natural outline.
5. The time cost for circle growth ought to be less than 2 seconds on average so as to provide immediate feedback.

6.3.2 Workflow of Circle Growth

Figure 6.5 shows the workflow of our circle growth method. It consists of two stages: locating new circles and establishing their size and suitability.

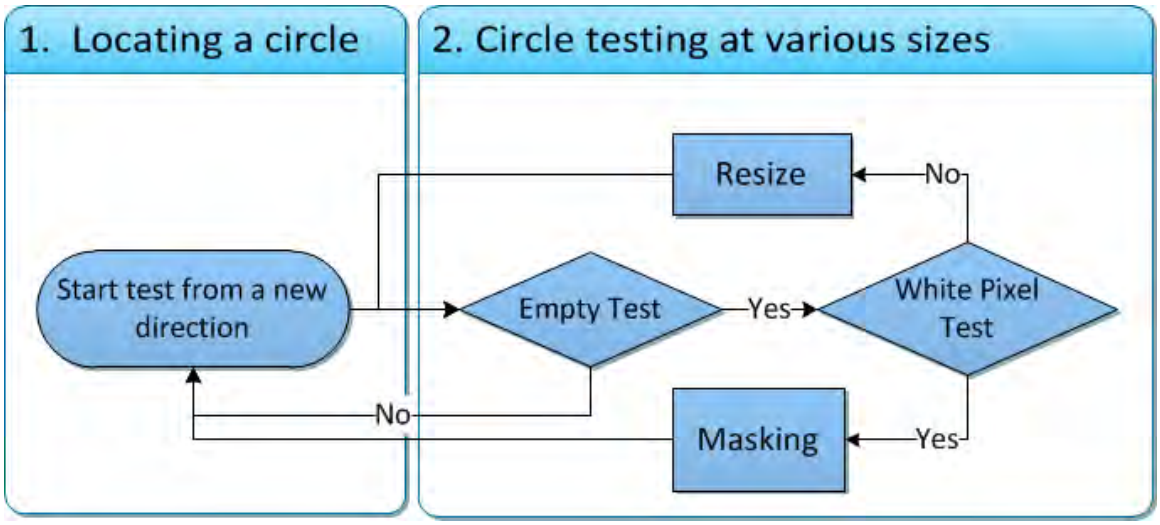


Figure 6.5: The workflow of our circle growth process contains two stages. The first stage defines where the new circle grows from and the second stage works on how large the circle grows.

When positioning new circles, we differentiate between the first circle and the subsequent ones. There are many ways to initialize the location of the first circle in the footprint, such as stochastically choosing any white pixel or selecting the middle white pixel along the drawn stroke. We choose the latter approach, which works well in practice.

The second case involves locating a new circle adjacent to an existing one. We sequentially test potential circle centres in 16 directions radiating evenly from the current circle. At each new location, the centre is kept constant while the radius is varied (stage 2 in Figure 6.5). We next undertake a pixel-level analysis to determine the proportion of white (cloud) pixels and empty (background or previously covered) pixels relative to the circle area. An *Empty Test* determines whether to discard the circle entirely because it overlaps the cloud boundary. If this passes, then a *White Pixel Test* is used to decide whether resizing the circle is required. This stage has a significant impact on overall performance, as the pixel analysis involves a large number of array traversals. To reduce this, both the *Empty Test* and later *Masking* procedure allow for early termination of the testing loop.

6.3.3 Pixel Analysis in Circle Growth

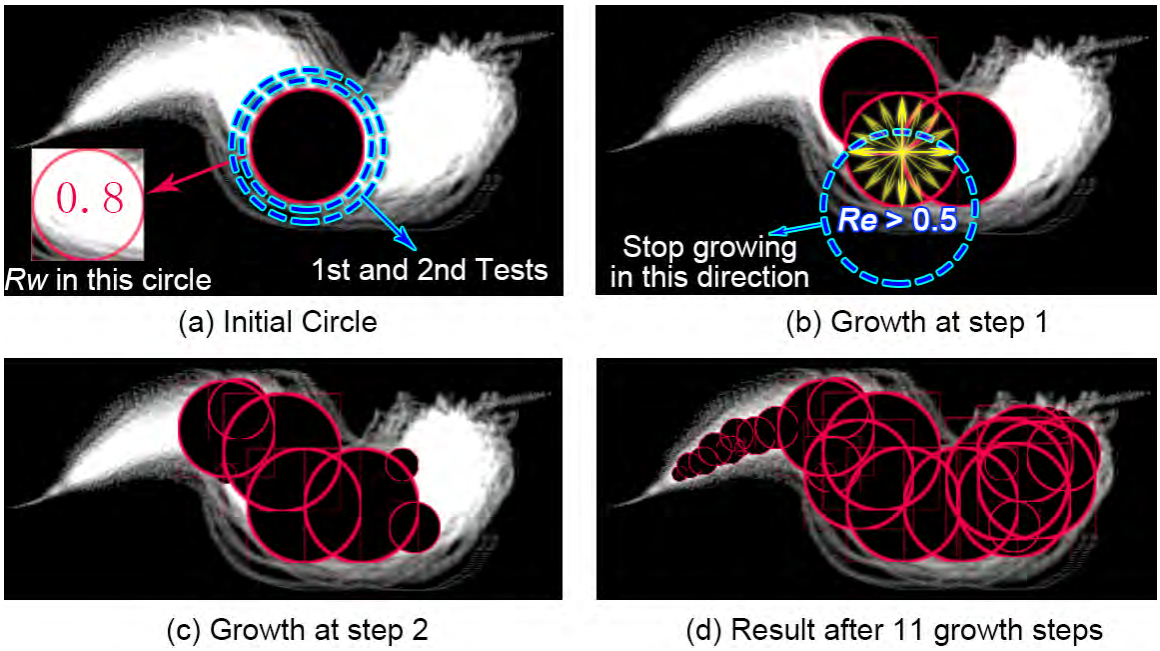


Figure 6.6: Stages in populating a footprint with circles. Circles are seeded adjacent to existing circles and expanded based on the proportion of white cloud pixels that they contain.

Figure 6.6 demonstrates the four phases of circle growth for a given footprint. Given an initial location within the footprint, a circular area, illustrated by the outer blue ring in Figure 6.6(a), is initialized for the collection of pixel information. In this case, the diameter of the circle starts at 128 pixels. Three metrics are calculated for each circle:

1. N_p — the total number of pixels covered by the circle;
2. N_w — count of white pixels, for which all the colour channels are 1.0f.
3. N_e — count of empty pixels, for which all the colour channels are 0.0f.

These are used to calculate $R_w = N_w/N_p$, the proportion of white pixels, and $R_e = N_e/N_p$, the proportion of empty pixels, which are used in the corresponding test conditions. R_w is the value used to determine the success of circle testing. R_e establishes that testing is taking place on an empty region, which can be skipped to enhance performance.

In Figure 6.6, R_w is set to 0.8 and the R_e is set to 0.5. Therefore, if 50% of the pixels in the circle under consideration are empty, this direction is immediately dropped from consideration and no further testing with smaller radii is undertaken. Instead the next radial direction is considered.

On the other hand, if 80% of the pixels are white then the circle is accepted. The location and the radius of the circle are pushed to an array for the later use in the 3D mapping stage. The area covered by the circle is also masked with empty values *float4(0,0,0,0)* so as to prevent subsequent duplicate tests of the same region. Figure 6.6 (a) shows that in this case it takes three attempts to succeed at the circle test. After each white pixel test failure, the diameter is reduced by 8 pixels, illustrated by blue dotted rings. The third test at a diameter of 112 pixels meets the 80% threshold and the area is masked in black.

In the next step, new candidates are distributed along 16 directions from the centre of the accepted circle (shown with yellow arrows in Figure 6.6(b)). Their centers are located along the perimeter of the accepted circle. The blue dotted ring in Figure 6.6(b) refers to the first test in the downward direction, but this fails as the R_e value is more than 50%. Thus, further circle tests in this direction are cancelled. At the end of this step, two circles have been successfully tested and masked as black circles.

This process continues until no candidate circles remain. Figure 6.6(c)(d) shows the growth at step 2 and the final result from this footprint. There are no further circles grown after 11 steps in this footprint.

This formation of the generated circles is satisfactory. Only 26 overlapping circles are required to fill most of the footprint, which satisfies the first three aims of our circle growth goals in section 6.3.1. In addition, these circles visually overlap each other in a stochastic fasion with a radius varying between 128 and 4 pixels. This accords with aim 4. Apart from performance considerations at this point, we consider this circle growth method as providing good quality in forming the body of a given footprint.

Algorithm 6 shows the pseudo code for our circle growth method. It involves four nested loops and a function of 2D array traversals in line 12. With the exception of the loop for testing 16 directions in line 7, the number of iterations for other loops varies based on previous growth results. This means these nested loops are difficult to

Algorithm 6 Circle Growth Method with four nested loops.

```
1: procedure CircleGrowth
2:     do                                          ▷ Starts a new step for Circle Growth
3:         NumCircleAtCrtStep = 0;
4:         PrvCircles[] = GetCirclesFromPreviousGrowth();
5:         for i ← 1, n = PrvCircles.size() do
6:             C_p = PrvCircles[i].Center; R_p = PrvCircles[i].Radius;
7:             for j ← 1, n = 16 do                ▷ Tests in 16 directions
8:                 Radius = 128;
9:                 Center = C_p + Direction[j] * R_p;
10:                do                              ▷ Pixel analysis at different radius
11:                    R_w = 0; R_e = 0;
12:                    GetFillRate(Radius, Center, &R_w, &R_e);
13:                    if R_e > 50 then            ▷ Skip testing in this direction
14:                        break;
15:                    end if
16:                    if R_w > 80 then            ▷ Test succeed
17:                        ArrayCircles.push(Radius, Center);
18:                        NumCircleAtCrtStep++;
19:                    end if
20:                    Radius−= 4;
21:                while Radius > 8
22:            end for
23:        end for
24:    while NumCircleAtCrtStep > 0                ▷ Keep growing until 0 new circles
25: end procedure
```

unroll and parallelize, which limits the implementation of this part to the CPU only. Many other essential functions such as memory transfer and pixel masking, are left out in this pseudo code since their locations impact on the final performance as it relates to the GPU implementation for the function of line 12. We will discuss the memory transfer and masking implementation for CPU and GPU separately in section 6.4.

6.3.4 Other Circle Growth Results

Figure 6.7 illustrates the effect of the initial diameter and FillRate on the number of circles generated. Strokes in Figure 6.7(a)(b)(c) have the same FillRate value:

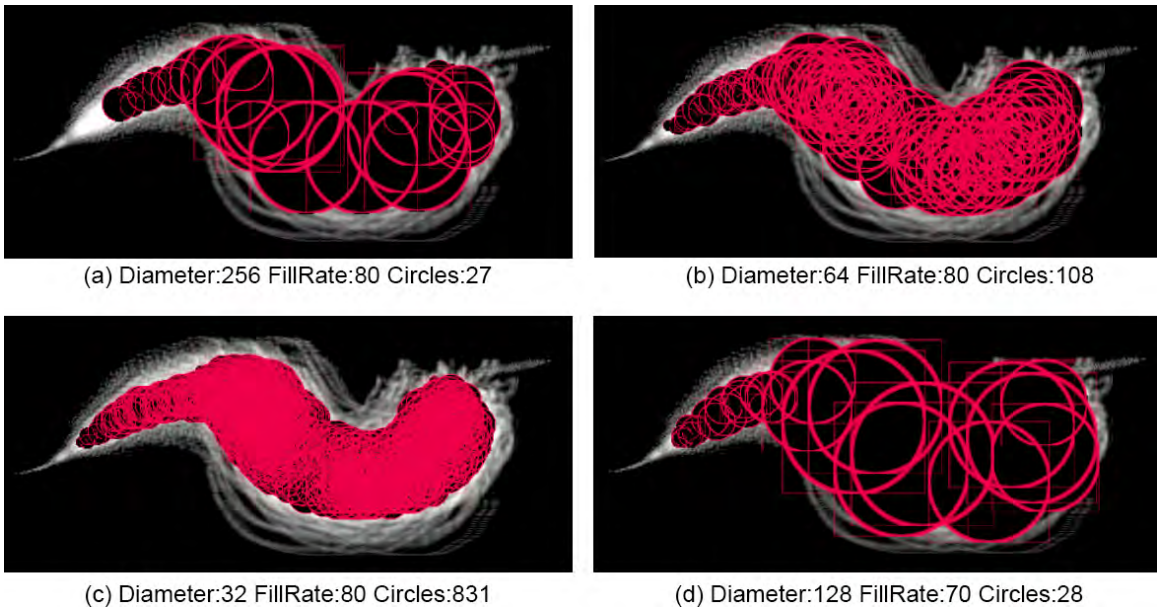


Figure 6.7: Various circle growth results using different initial diameters and FillRate parameters. (a)(b)(c) Same FillRate with various diameters produces different number of circles. (d) Same diameter as Figure 6.6 with various FillRates also leads to different number of generated circles.

80%. They produce different numbers of circles due to the various initial diameters. Figure 6.7(d) starts with the same diameter as our running example, 128 pixels, but the R_w value is set to 70%. Compared to Figure 6.6(4), the circles are larger which extends the coverage to the very edge of the footprint. This may fail to maintain the integrity of all boundary features as some edge details are lost in the later edge mapping stage.

Circle Growth on various Strokes

Figure 6.8 demonstrates the results of circle generation from four strokes using the same fixed parameters. Circles are generated along the flow of the stroke, which is represented by a green-dotted line. The circle sizes change according to the thickness of the footprint. It is clear that this method is able to sensibly fill most of the example footprints.

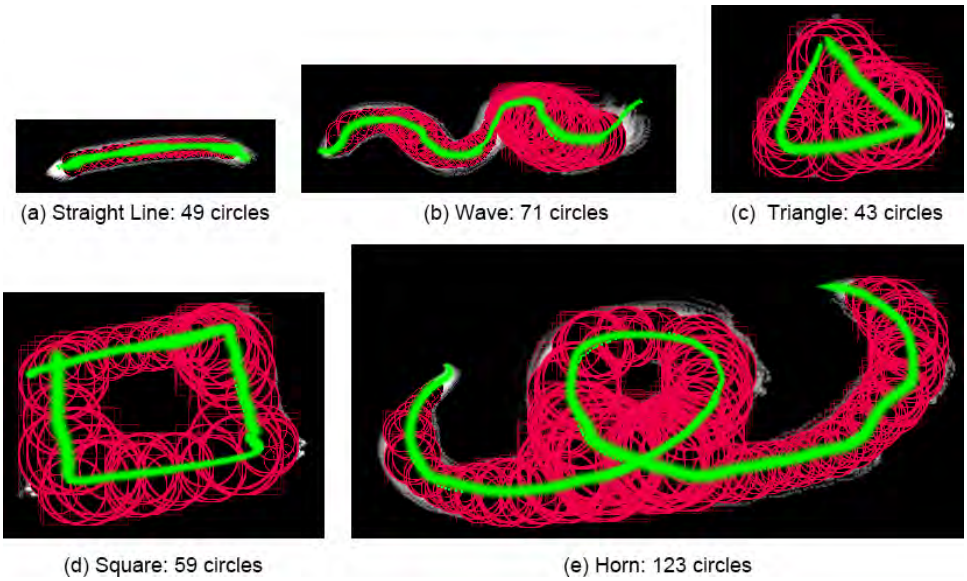


Figure 6.8: Various results using the same initial diameters (128) and FillRate (80) during circle growth. Circles are generated follows the path of the stroke drawing, which is highlighted in green.

Large Footprint with Crossed Stroke Flows

Figure 6.9 shows a footprint in the shape of a heart. The green line indicates this footprint is created through a complex brush flow. The result confirms that our method is able to grow the circles in various directions so as to approximate the shape from a large footprint.

Figure 6.9: A heart shaped footprint with complex crossed stroke flows generates 163 circles.

6.4 Implementation Using CPU and GPU

Although the *Circle Growth Method* only executes once the user finishes a stroke, it is vital for this process to complete quickly. A user's experience can be easily damaged if he or she has to pause for more than 10 seconds after each stroke is drawn. We tested several optimizations to enhance performance. Most importantly, we examined optimizations for the circle growth method for both CPU and GPU in order to identify the best hardware device on which to run this process.

6.4.1 Motivation for Comparison Test

Here we list the major considerations that motivated us to implement a performance test on both units.

- For CPU units:
  1. It is less efficient to process textures on CPU than GPU since no exclusive units for texture access and sampling are provided on CPU.
  2. The data transfers between GPU and CPU cost extra performance. The CPU has to retrieve the footprint texture from GPU memory so as to analyse the texture data.
- For GPU units:
  1. The number of pixels required for parallel reduction operations is fairly limited—in most cases less than 16k (128×128). Within this limit, the study from Lungu etal [75] shows that the CPU is more efficient in accumulation operations than the GPU.
  2. Once the GPU is involved in the circle growth stage, it is difficult to start a new thread for the Compute Shader (CS) process while the current DirectX 11 thread is still busy with volumetric rendering. If the circle generation and rendering are implemented sequentially on one thread, rendering will stall while the CS starts circle growth. This stall may occur frequently for each stroke drawing, which can severely affect a user's experience.

6.4.2 The Performance Impact of Texture Traversal

We conducted tests for both GPU and CPU implementations. Our testing device is a laptop with Intel i7-3610QM and Nvidia GT650M graphics card, which supports DirectX 11 with Compute Shader 5.0. The resolution of the rendering window is 1280×768. The primary difference between the CPU and GPU implementations is the control flow of the function $GetFillRate()$, (Algorithm 6 line 12), which traverses a 2D texture and processes a data reduction operation [46, 121].

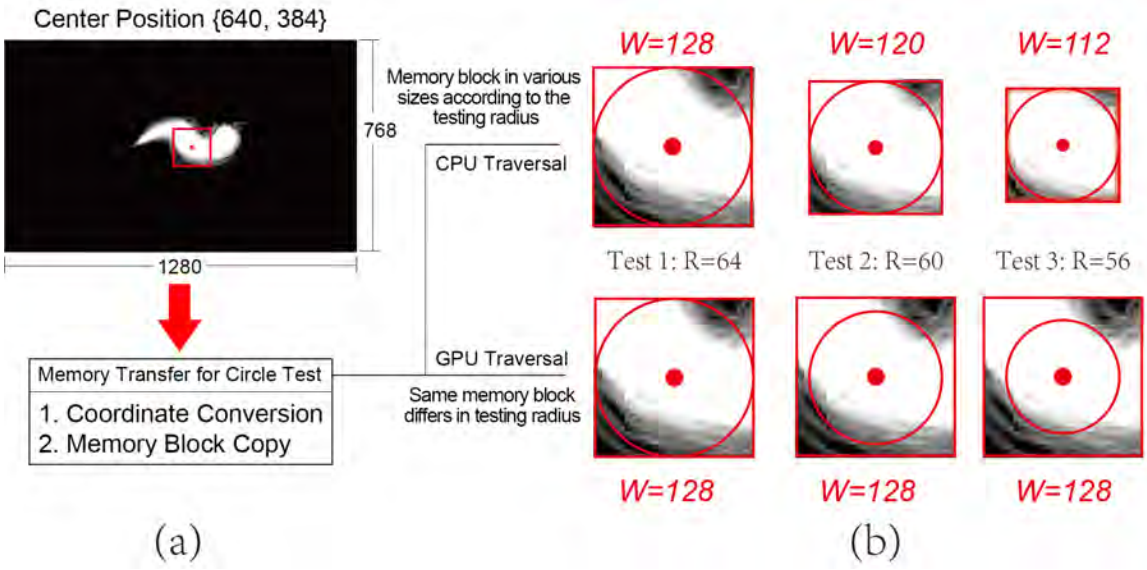


Figure 6.10: Different memory transfer strategies in texture traversal using CPU and GPU. (a) Once a location from the footprint starts circle testing, the corresponding memory block is trimmed and converted so as to be quickly transferred for traversal. (b) The CPU and GPU traversal use various memory copies with regard to the size of the memory block. CPU implements various sizes in each test while GPU remains the same.

Figure 6.10 shows these different memory transfer strategies with CPU and GPU implementations for texture traversal. Instead of traversing the entire size of the footprint texture, which leads to a large number of iterations, 960k (1280×768), a small block with regard to the testing circle is traversed at each step so as to reduce the number of processed pixels by at least 98%. In Figure 6.10, step(a) demonstrates this block clipping stage. It is important to note that the centres of the testing circles, C_{circle}, may vary from the center of the clipped blocks. This

situation occurs when the testing circle is close to the edge of the footprint while its radius, R_{circle}, is larger than the distance between the centre and edge of the footprint: $R_{circle} >= Dis(C_{circle} - Edge)$. Therefore, step(a) involves a coordinate transform which produces a correct centre position in the 2D memory block with new width and height variables. After that, Figure 6.10(b) shows two strategies that transfer memory blocks in both the CPU and GPU implementations. In the CPU version, the size of the memory block changes with the radius of the testing circle. However, the GPU implementation uses a fixed memory block size due to a hardware-based texture access feature. The different traversals on both the CPU and GPU versions are discussed in following sections.

6.4.3 CPU Traversal

In CPU mode, the first task is to copy the entire footprint from GPU memory to system memory once the circle growth starts. This prevents frequent GPU accesses in the later circle testing stage since small memory blocks can be efficiently duplicated from the system memory. The size of the memory block varies according to the radius of the testing circle. As the radius reduces by 4 pixels for each test, the top row of the Figure 6.10(b) demonstrates a series of sequential blocks with 8 pixels difference in width. This enables the minimum number of traversals for the pixel analysis function, $GetFillRateUsingCPU()$.

Algorithm 7 lists the pseudo codes for the CPU implementation of the function GetFillRate(). The traversal through the PixelArray is serialized. The total number of traversals varies depending on the radius of testing circles. In this case, the maximum number of traversals is 16k and the minimum is 16, if the radius of the testing circle is 4 pixels. The generated variables R_e and R_w are used for $EmptyTest$ and $SuccessTest$ in line 15 and 16 of Algorithm 7.

Algorithm 7 The CPU implementation of function GetFillRate().

```
1: procedure GetFillRate(Radius, Center, &R_w, &R_e)
2:     CoordinatesConvert(Radius, Center, &NewWidth, &NewHeight, &NewCenter);
3:     MemoryCopy(&PixelArray, &SrcTexture, NewWidth * NewHeight);
4:     NumPixelsInCircle = 0; NumWhites = 0; NumEmpty = 0;
5:     for y ← 0, n = NewHeight do
6:         for x ← 0, n = NewWidth do
7:             if DistanceFromCrtPixelToNewCenter(x, y) <= Radius then
8:                 NumPixelsInCircle + +;
9:                 Color ← PixelArray[y * NewWidth + x];
10:                NumWhites+ = (Color == White)?1 : 0;
11:                NumEmpty+ = (Color == Empty)?1 : 0;
12:            end if
13:        end for
14:    end for
15:    R_w = NumWhites/NumPixelsInCircle;
16:    R_e = NumEmpty/NumPixelsInCircle;
17: end procedure
```

6.4.4 Parallel reduction in the GPU implementation

The GPU implementation uses another method, as textures are accessed differently. In the CPU implementation, memory blocks can be managed at any requested size. Unfortunately, for performance reasons DirectX 11 works better on textures that are a multiple of 64. Processing arbitrary sizes of textures can lead to unexpected results, as shown in Figure 6.11.

To avoid this situation, the size of the testing memory blocks on the GPU is fixed to 128 by 128 pixels, as shown in the lower row of Figure 6.10(b). Although this does not minimize the number of traversals, it only requires one memory copy operation prior to the tests. Thus, the subsequent tests work on the same memory block located in GPU's texture registers. A total of 16k (128 by 128) threads in the Compute Shader are used to analyse this texture in a parallel manner. Different radii required by different tests are sent to the Compute Shader through a constant buffer.

Algorithm 7 explains how the reduction operation is implemented on the CPU. The testing block is serialized by two 'for' loops so that each pixel can be analysed in a

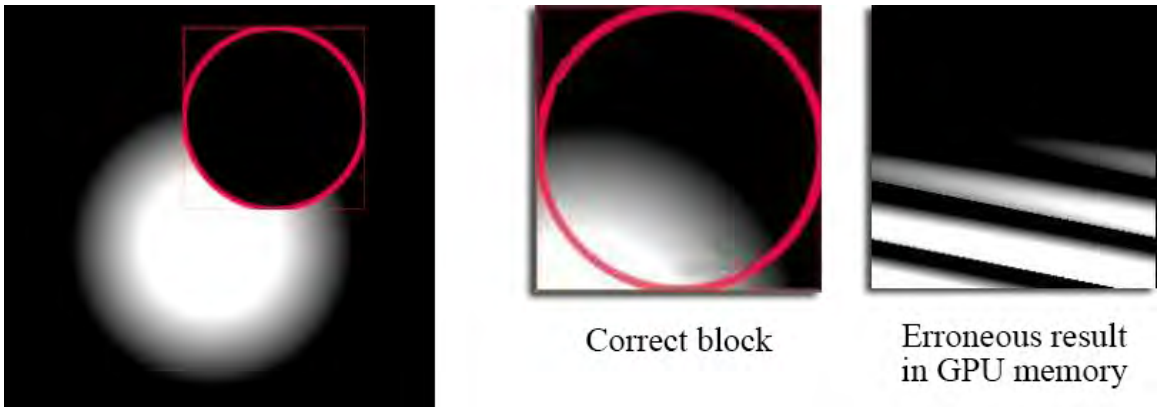


Figure 6.11: The representation of a 203×203 texture in GPU memory. Note the unexpected texel distribution in GPU memory for a texture with the size other than multiples of 64.

sequential order. However, a parallel reduction operation is not easy to implement efficiently on a GPU. It is considered to be one of the bottlenecks in GPGPU algorithms due to the large size of the datasets and difficulty in direct memory access [50]. With the capabilities of new GPGPU units such as CUDA and Computer Shader, Sengupta *et al.* [104] and Harris [45] show how to implement a fast reduction algorithms on GPGPUs. In order to get well-optimized performance from a dataset with more than 4M elements, Harris [45] explains how to use optimization strategies such as sequential addressing and a complete unroll, etc. In our case, the size of the dataset is 16k. There are three groups of variables (N_p, N_w, N_e) collected from this dataset. So, both the scale and number of the reduction objects are different from the above mentioned studies. Therefore, we implement an easy to understand approach by leveraging the atomic functions provided in Compute Shader 5.0 supported GPUs.

Figure 6.12 demonstrates our pixel analysis procedure for the GPU. Figure 6.12(a) is a testing block, which has been shown previously in Figure 6.10(b). In this circle test, the size of this testing block is 128 by 128 pixels, and it is located in the GPU's memory. It contains a circle with a radius of 56 pixels. As atomic functions only work well with groupshared memory and the size of groupshared memory has a limitation of 16KB per thread group, we dispatch a total of 16k threads (128×128) into 256 groups. In this way, each group deals with 64 threads (8×8) as shown in Figure 6.12(b). The 64 threads inside a group can 'InterlockedAdd' three variables N_p, N_w, N_e to a series

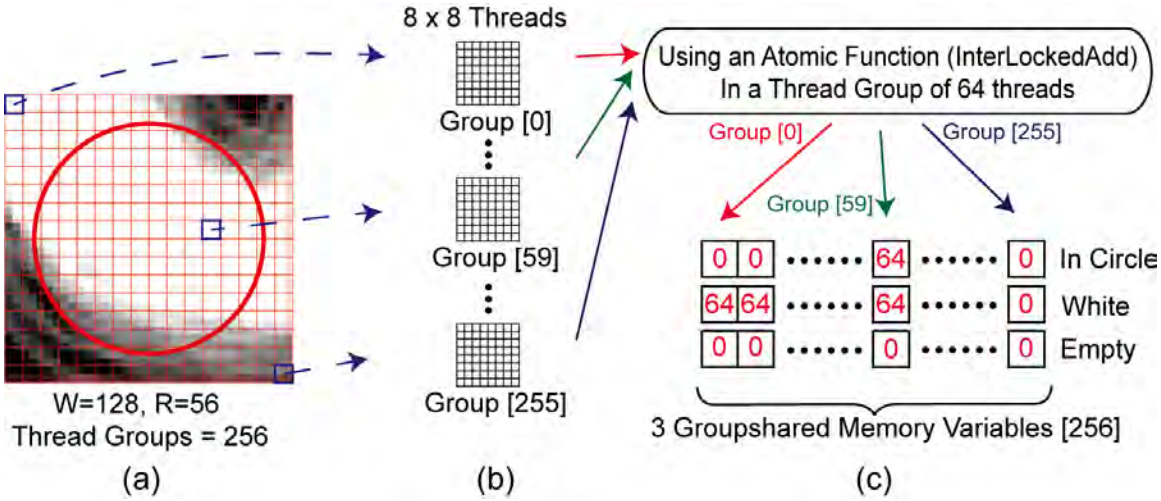


Figure 6.12: Our procedures for pixel analysis using the GPU. Note that all pixels in group 255 are in varying levels of gray color, thus no white and empty are counted. This flow only represents the first dispatch call. A second dispatch is called to implement a full reduction of the three sets of arrays.

of corresponding arrays which are located in groupshared memory. The index of the array is correlated to the group index as shown in Figure 6.12(c). This minimizes the running size of groupshared memory to 3k in order to enhance performance.

For example, group 0 in Figure 6.12 is the first group from the top left of the testing circle. All its pixels are white and none of them is in the circle. Thus, after the atomic functions, the total number of pixels in the circle, which is 0 in this case, is saved in the first element of the 'In Circle' array. The total number of white pixels (64) is saved in the first element of the 'White' array. Both group 59 and the last group, 255, are processed with the same strategy, as are on groups inbetween.

The results from this dispatch call are three sets of 256 element arrays representing values of N_p, N_w, N_e for each group. Then, a second dispatch is called to reduce the three sets of arrays into three variables only. In this dispatch, we follow the optimizations suggested by Harris *et al.* [45].

Algorithm 8 lists the pseudo code of our GPU implementation. Similar to the CPU implementation, the GPU implementation also involves a coordinate conversion and memory copy which is executed in the first line. The fourth and fifth lines represent two dispatch calls to the GPU. The final reduction results are mapped out

Algorithm 8 The GPU implementation of function GetFillRate().

```
1: MemoryCopyOnGPU(&NewTexture, &SrcTexture, NewWidth * NewHeight);
2: procedure GetFillRateUsingGPU(Radius, Center, &R_w, &R_e)
3:     UpdateToComputeShaderWithNewRadius();
4:     csForFillRate.Dispatch(16, 16, 1);
5:     csReduction.Dispatch(1, 1, 1);
6:     CSMappingOut(&N_p, &N_w, &N_e);
7:     R_w = N_w/N_p;
8:     R_e = N_e/N_p;
9: end procedure
```

from GPU to CPU in line 6 so as to calculate the R_w and R_e for growth determination.

Algorithm 9 The HLSL of Compute Shader CSGetFillRate.

```
1: groupshared int accumCircle[256],accumOne[256],accumEmpty[256];
2: [numthreads(8, 8, 1)]
3: procedure CSGetFillRate(Gid, GI, DTid)
4:     dis= length(DTid.xy-f2Center); iCircle=(dis<=Radius)?1:0;
5:     iOne= (iCircle==1 && tex[DTid.xy].a==1)?1:0;
6:     iEmpty= (iCircle==1 && tex[DTid.xy].a==0)?1:0;
7:     index=Gid.y*16+Gid.x;
8:     GroupMemoryBarrierWithGroupSync();
9:     InterlockedAdd(accumCircle[index],iCircle);
10:    InterlockedAdd(accumOne[index],iOne);
11:    InterlockedAdd(accumEmpty[index],iEmpty);
12:    GroupMemoryBarrierWithGroupSync();
13:    if GI == 0 then                                          ▷
14:        stbOut[index].uCircleCount=accumCircle[index];
15:        stbOut[index].uOneCount=accumOne[index];
16:        stbOut[index].uEmptyCount=accumEmpty[index];
17:    end if
18: end procedure
```

Algorithm 9 lists the pseudo code of the HLSL implementation for the first dispatch call. It starts with a declaration of three sets of groupshared memory. Each element of the array is correlated to the group index, which starts the reduction process from line 9. Note that although the strategy in this pseudo code works well

in our case, it still has room to improve since the use of 'InterlockedAdd' may cause errors if the number of threads grows.

6.4.5 CPU and GPU results

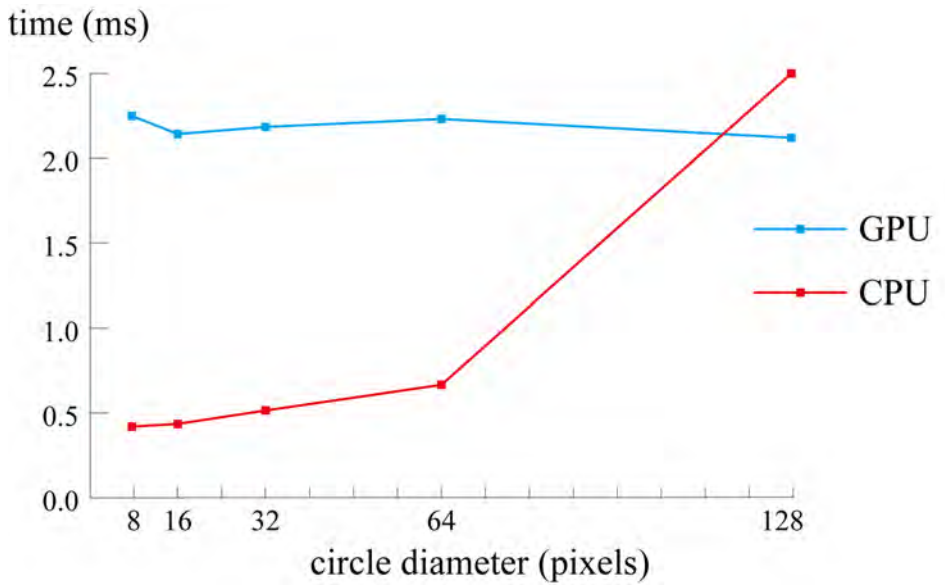


Figure 6.13: Performance comparison between CPU and GPU implementations of one circle test. The x axis represents the diameter of the testing circle and the y axis is the time cost in seconds.

In section 6.4.1, we listed two assumed disadvantages of using a CPU implementation, which are low efficiency in texture processing and extra memory transfer. Our comparison results from Figure 6.13 show that, in fact, the GPU implementation has no advantage in reduction operations compared to the CPU when the size of the texture is less than 128 pixels. This result also accords with the study of Harris *et al.* [45] and Lungu *et al.* [75].

Secondly, the data transfer between GPU and CPU only occurs once. It is executed before the circle growth starts with the total size of the footprint as 16M in our case. The time cost on data transfer (less then 10 ms) is negligible compared to the total growth time, which usually takes more than a second.

Therefore, we focus further optimization of the circle growth algorithm on the CPU implementation.

6.4.6 Further Optimization and Performance

Based on the previous comparison results, two new methods of CPU-based optimization are developed to improve the generation of circles.

A binary search method is implemented to reduce the number of total attempts for circle tests. Previously, the maximum number of test attempts in one direction was 31 as the radius for each attempt decreased by 4 pixels from 128 to 4. The binary search method, which has worst case $O(logn)$, limits the number of attempts to 6. With the help of masking area identification, the number of the attempts in each direction is from 2 to 4. This reduces the total number of attempts for circle growth by at least 87% and increases the performance significantly.

Another method of optimization is to use OpenMP [88] to parallelize the circle growth on multiple threads provided by the CPU. This also leads to a large performance increase.

Table 6.1: Full circle growth performance results using different diameters.

Diameters	Generated Circles	Growth Steps	Attempts	Time Cost
256	27	8	3511	0.2737 s
196	22	9	2660	0.1827 s
128	26	11	2992	0.2445 s
96	35	10	3723	0.2002 s
64	108	13	11264	0.5805 s
32	831	17	77347	20.4415 s

Table 6.1 shows the results of the optimized performance tests using different diameters. The footprint is the same one we presented in Figure 6.10. Some visual results of circle growth using various diameters such as 256, 64 and 32 were shown previously in Figure 6.7 of section 6.3.3. We also noted that we were satisfied with the visual results, useing a diameter of 128 and $R_w = 0.8$. From table 6.1, we observe that performance is satisfactory: we can generate 26 new circles in only 0.2445 seconds. This facilitates our requirement for interactivity.

From a performance point of view, other diameter results, with the exception of a diameter of 32 are also acceptable: the time costs using diameters from 256 to 96

are relatively flat at 0.2 seconds. Small diameters such as 64 and 32 pixels generate many more circles and circle test attempts. The number of total attempts, which easily exceeds ten thousand, significantly impedes performance. The waiting time of 20 seconds using a diameter of 32 is unacceptable.
Based on the above, we suggest using diameters of approximately 128 pixels to initialize the circle test in our circle growth method. This size not only provides a good result that fills most of the footprint body with a limited number of overlapped circles, but also maintains the circle growth at interactive update rates.

6.4.7 Circle Growth Conclusion

Our circle growth method aims to encapsulate the footprint using overlapped circles. The generated circles have varying radii so as to produce a randomized appearance. We implemented this method on both the CPU and GPU with a view to optimizing the performance for interaction purposes. Results indicate that the CPU implementation provides the best performance. Once the circles have been generated, they are mapped as spheres into the 3D world.

6.5 Sphere Mapping

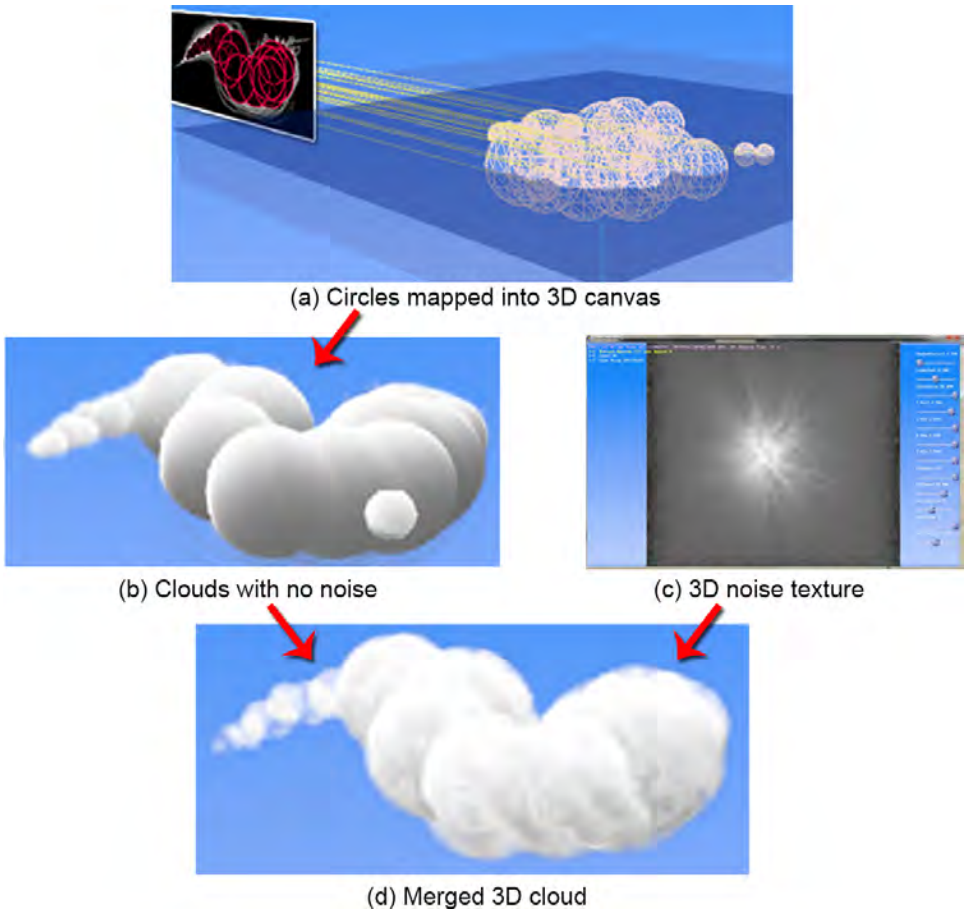


Figure 6.14: The workflow of sphere mapping. (a) Circles are mapped into a 3D canvas, which generate a volumetric cloud (b) in the 3D world with smooth surfaces. This smooth 3D cloud is merged with a pre-generated 3D noise texture (c), which produces a cloud (d) with a realistically irregular surface.

The second stage in our *Hybrid Mapping Method* is the sphere mapping. Figure 6.14 demonstrates how the generated circles are mapped into the 3D world with a noisy surface effect. In Chapter 3 section 3.4, we describe the setting of camera direction and the use of a 3D canvas in our cloud modelling system. Before the user starts painting a stroke, it is necessary to define a 3D canvas with values for altitude and thickness. The camera must also be adjusted to provide the desired view direction towards the canvas. In the stage of sphere mapping, the locations of the mapped

spheres are affected by both the camera setting and the canvas altitude.
Figure 6.14(a) elaborates this sphere mapping process. The semi-transparent grid represents the 3D canvas with its altitude represented as a dark-blue plane. The yellow lines indicate the mappings from the circles along the camera's view direction, which intersect with the altitude plane. These intersections define the location of the spheres in the 3D world. Figure 6.15 shows the top view of the same mapping, which indicates the various depths of mapped spheres in z coordinates.

Figure 6.15: Sphere mapping from the top view. Yellow rays indicate the sphere mapping directions emitted from the same painting view, but map the sphere at various depths.

Once the spheres are mapped, texels within the boundary of these 3D spheres are set to a value of 1.0 via the Compute Shader. The result of a mapped cloud dataset is shown in Figure 6.14(b). It is easy to see that such smooth spheres are far from a realistic depiction of clouds.
Our strategy is to merge the resulting volume with a pre-generated 3D noise texture (Figure 6.14(c)). The 3D noise texture is produced by Simplex Noise functions [92], which are only generated when the system is launched. Accumulating the mapped

spheres and the 3D noise texture is implemented in a single dispatch using the Compute Shader.

The outcome is a 3D cloud with a noise-modulated surface as shown in Figure 6.14(d). However, this fails to match the brush footprint exactly because there are cloud pixels remaining that fall below the circle fill threshold. In the next section, we present our solution to this problem.

6.6 Edge Mapping

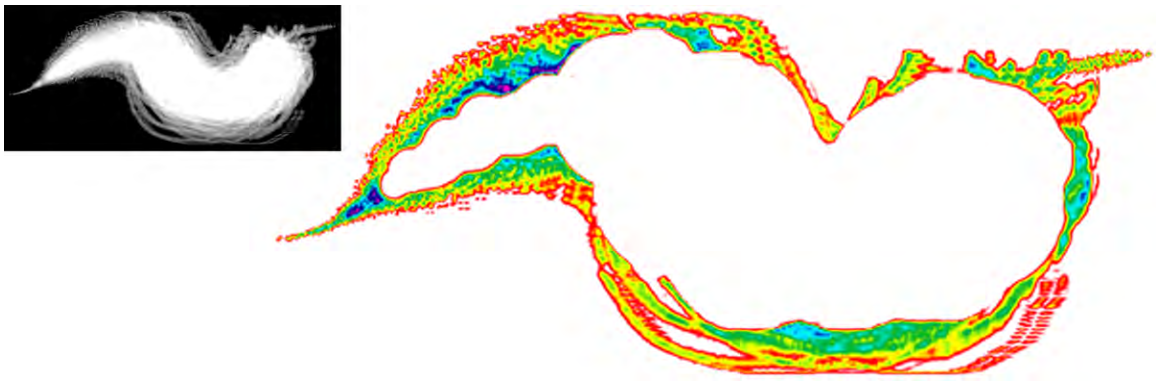


Figure 6.16: The edge details left out from circle growth. The red pixels, whose density values are lower than 0.4, preserve the high-frequency features from the boundary of the brushwork. The yellow, green and blue pixels indicate density values between 0.4 and 0.9.

Although using the 3D noise texture provides a rough surface effect, many boundary features from the footprint remain excluded after the data mapping stage. Thus, after all circles have been generated and mapped into the 3D canvas, the remaining non-empty pixel positions from the masked footprint are collected. This is illustrated in the image on the right of Figure 6.16, which contains 17,325 non-empty pixels. These pixels carry the high-frequency boundary features of the brushwork and it is important to capture these details in the final cloud.

6.6.1 Canvas-Based Mapping

In the edge mapping stage, we map all boundary pixels to the 3D canvas in order to preserve comprehensive footprint features such as hairy effects, sharp edges and scattered dots, etc. Each pixel is mapped from the same view direction as the one used in the sphere mapping stage. We also implement this using the Compute Shader so that the 3D dataset can be updated directly in the GPU's memory. However, unlike the sphere mapping method, which intersects the mapping rays with the altitude plane of the 3D canvas, it is difficult to determine where the boundary pixels should be mapped.

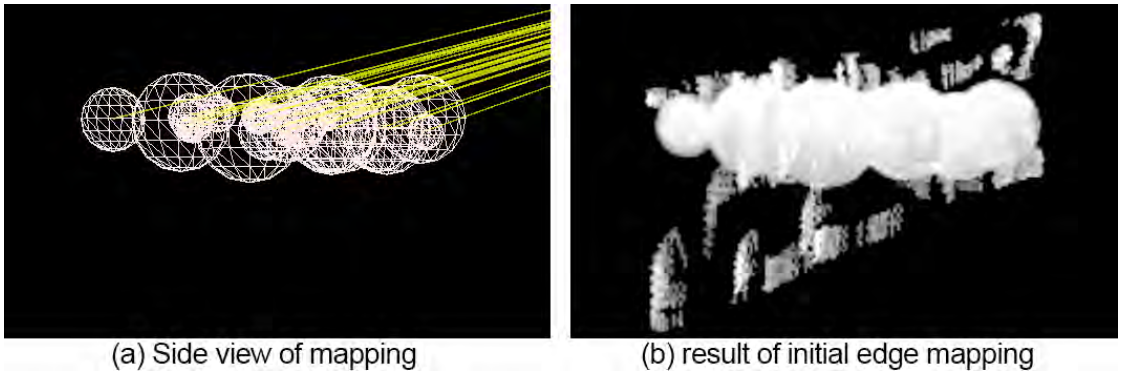


Figure 6.17: Initial edge mapping that locates the mapped voxels along the rays clipped by the 3D canvas. (a) The side view indicates the direction of mapping is from top right to the middle. (b) The poor result from our initial edge mapping.

Initially, we planned to distribute the boundary features around the entire cloud. Figure 6.17 shows our initial edge mapping strategy. This strategy first clips the mapping rays with the 3D canvas. Each boundary pixel is then scattered along its clipped mapping ray.

Figure 6.17(b) shows the result of the canvas-based edge mapping, which did not yield the results we expected. The mapped boundary pixels, particularly those at the beginning and the end of the mapping rays, are distant from the cloud body and distributed in a straight line. This regularity produces an obvious artifact in the generated cloud. Thus, instead of feature extension to the entire cloud, a more practical strategy is to apply the boundary features to certain regions that are more easily perceived from the view direction.

6.6.2 Sphere-based Mapping

Figure 6.18(b) demonstrates the same side view of the same cloud spheres using our sphere-based edge mapping method. The cloud spheres on the right are located closer to the viewers than the left ones. Surface details from these closer clouds are normally easier to perceive than the farther ones. Thus, our implementation of the edge mapping focuses on these spheres and leaves other parts with the general rough surface representation generated from the 3D noise texture.

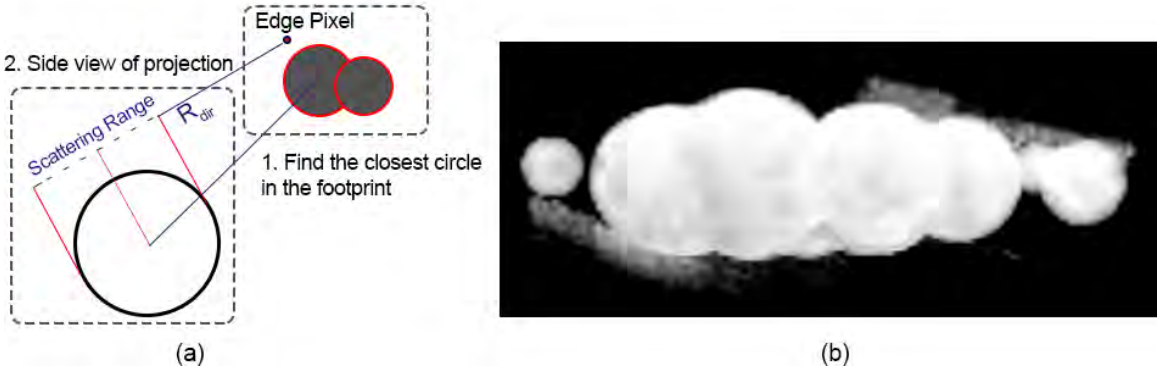


Figure 6.18: Sphere-based edge mapping method. (a) The edge mapping has two steps: finding the closest circle and mapping to the orthogonal view plane of the 3D sphere. (b) The result from the side view indicates most top edge pixels are scattered along the right cloud, which is closer to the viewer. The bottom edge details are mapped to the left clouds, which are farther but unblocked from the viewers direction. This balances the distribution of edge mappings for the whole cloud.

Figure 6.18(a) shows the strategy of the sphere-based edge mapping. The mapping of each edge pixel requires two steps: 1. Find the pixel's closest circle in the footprint. 2. Map to the orthogonal view plane of the 3D sphere. The details are described as follows:

1. For each non-empty pixel, we traverse all the generated circles to find the closest circle. We thus collect the following values:
   (a) The minimum distance (D_{min}) between the pixel and the nearest circle.
   (b) 3D position (P_S) and radius (R_S) of the closest mapped sphere.
   (c) The viewpoint (P_{orig}) and the direction of view ray (R_{dir}), which passes from the viewpoint through the pixel's centre.

2. The position of the tangent intersection (P_t) is calculated based on the view ray, the location of the nearest sphere with radius of D_{min}. The final mapped positions P_f are then scattered around P_t in a Gaussian distribution with a variance smaller than R_S.

$$\begin{aligned} P_t &= TangentPoint(P_{orig} + R_{dir}, P_S, D_{min}) \\ P_f &= P_t + R_S * GaussianRand()/3.0f \end{aligned}$$

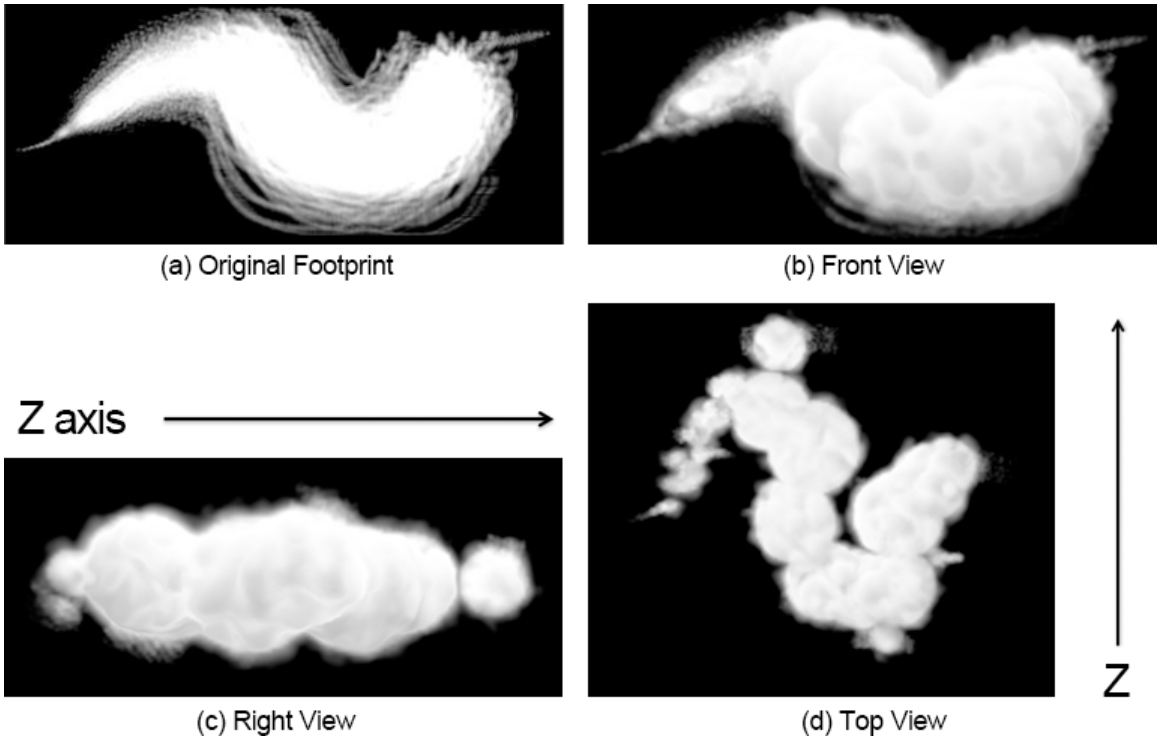


Figure 6.19: The result of hybrid mapping from three different views as compared to the original brush stroke. This 3D cloud looks realistic from all viewing directions, while preserving both the shape of the stroke body and its edge details.

Unlike the canvas-based mapping, the sphere-based method distributes the mapped voxels closely around the cloud spheres due to our closest circle finding strategy. Although the result in Figure 6.18(b) also indicates a certain degree of regularity, this

artifact can be largely removed by merging with the noise texture.

Figure 6.19 shows the outcome of our Hybrid Mapping Method, which combines both the previous noise surfaces and our sphere-based edge mapping method. Compared against the original footprint (shown in Figure 6.19(a)), the generated 3D cloud captures most of the footprint boundary features, including the sharp tips at both ends. Striations along the bottom curve are also preserved in the 3D cloud. The cloud also looks realistic from views other than the original painting perspective (Figure 6.19(c),(d)).

It is worth noting that this edge mapping method only captures the edge features from the painting view. It would be difficult to identify these features when the view direction changes substantially, as we have not implemented any reconstruction techniques that extend the features to the entire cloud surface. This feature extension technique will be explored in our future work.

6.7 Performance of HMM

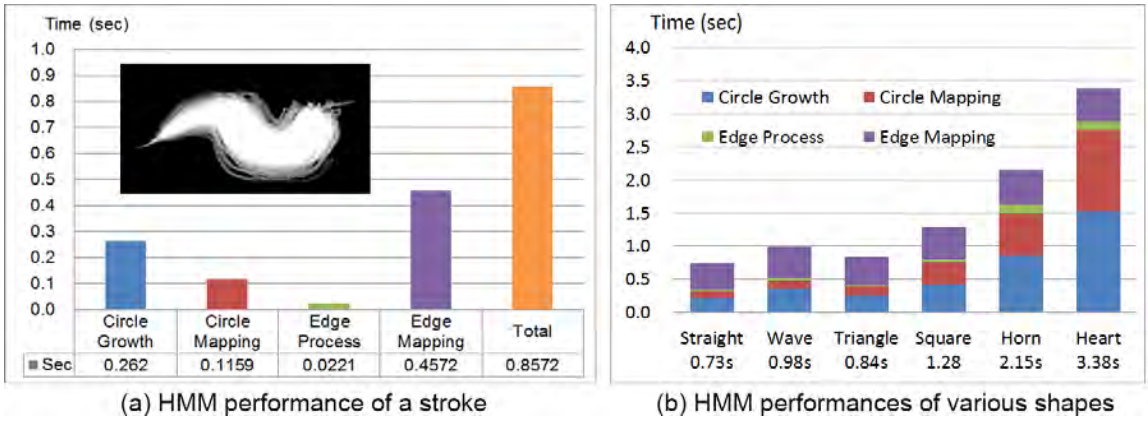


Figure 6.20: The HMM performances. (a) The HMM performance histogram of all stages in a stroke. (b) HMM performances of various shapes in Figure 6.8 and Figure 6.9.

For easy reference purposes, we have used the same footprint texture many times in explaining all the stages in our *Hybrid Mapping Method*. Figure 6.20(a) shows the performance results for all stages in the mapping of this footprint. The total mapping time in this instance was 0.86s on a laptop with i7-3610m (8GB RAM) and a GT

650M graphics card (2GB RAM).
The graph indicates the major performance costs are in the stages of circle growth and edge mapping. The circle growth stage is performed on the CPU, which involves a large number of texture traversals for pixel collection operations. The edge mapping is implemented on the Compute Shader, which merges the boundary features and the noise samples in a single dispatch call. Its performance is affected by the number of boundary pixels (17,325 pixels in this case) that causes massive structured buffer loading. In general, the total time cost of 0.86 seconds accords with our fifth aim described in section 6.3.1, which is less than 2 seconds so as to provide immediate feedback. Figure 6.20(b) shows the *HMM* performance of various shapes from Figure 6.8 and Figure 6.9. The stage distributions vary in these strokes due to different numbers of generated circles and edge pixels. However, the edge process is usually the fastest stage in *HMM*.
Figure 6.21 and Figure 6.22 reveal both the *HMM* results and their time cost with regard to the strokes used in Figure 6.8. With the implementation of our volumetric visualisation technique, the generated cloud data can be rendered in real-time with compelling visual quality. The time cost varies depending on the complexity of the stroke footprints. Although most simple strokes in Figure 6.21 require less than 2 seconds for the *HMM*, it is necessary to evaluate the acceptability of this time lag and the entire cloud modelling method by means of user evaluation.

6.8 Summary

In this chapter, we proposed a method for cloud generation based on the mapping of a 2D footprint. This *Hybrid Mapping Method (HMM)* involves three stages, which firstly produce 2D circles inside the footprint and then map the circles and their boundary features into a 3D canvas. The last stage merges the mapped cloud data with a 3D noise texture to enhance the cloud surface with a roughened appearance. We implemented and compared both CPU and GPU versions in order to determine optimized performance. The next step is to conduct an user evaluation for our cloud modelling system.

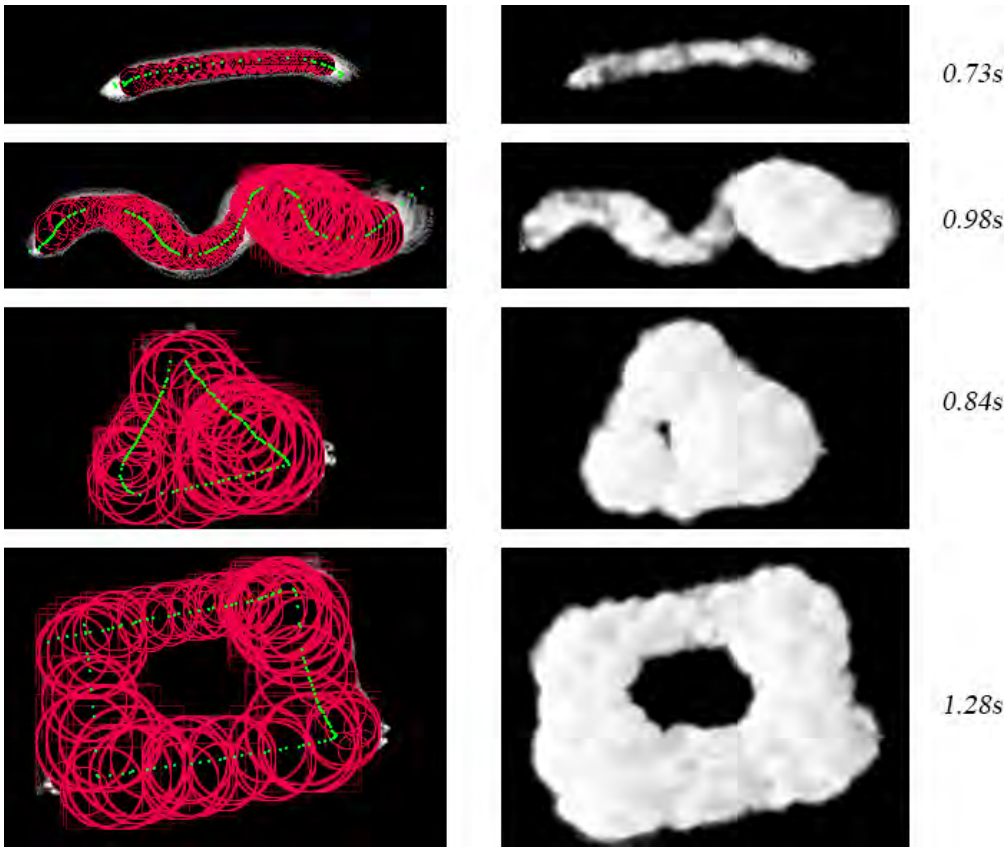


Figure 6.21: The HMM results for simple strokes that take less than 2 seconds.

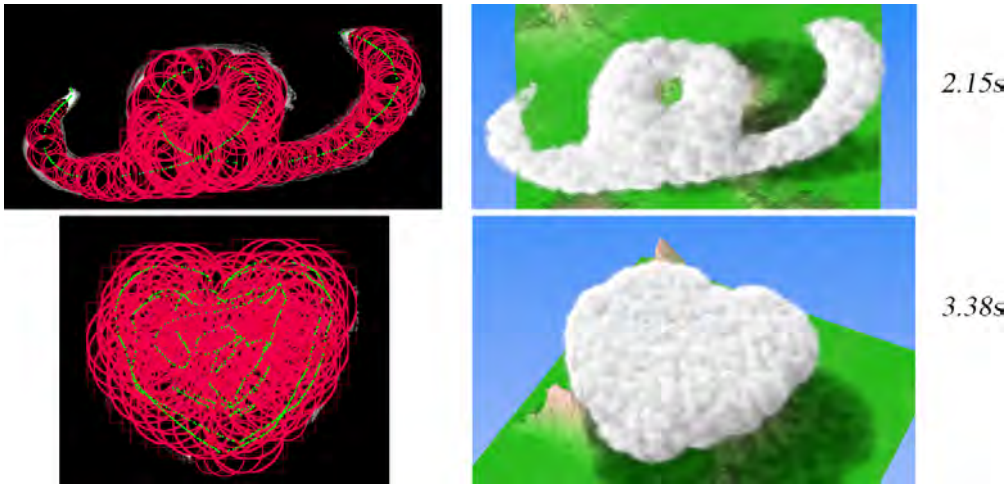


Figure 6.22: Strokes that produce heavy time lags from the HMM process.

Chapter 7

User Evaluation

Cloud visualisation, brush simulation and the cloud growth method are the three core modules in our cloud modelling system. These modules are managed through an integrated user interface with three modes: Camera, Canvas and Paint. Chapter 3 describes the flow of these modes, in which modellers are able to paint the cloud and visualize the result in an interactive fashion. A keyboard and a tablet pen are required as the input devices. In this combination, the pen is mainly used for painting with the keyboard executing the commands, such as switching between the three modes, etc. To the best of our knowledge, our work is the first to generate 3D clouds using a brush calligraphy system. As such, it is necessary to conduct a comprehensive user evaluation for this new interface in order to determine its effectiveness.

7.1 Evaluation Criteria

We want to learn if our brush calligraphy method is effective for generating compelling volumetric clouds. Therefore, a user evaluation is essential to assess the *Brush Calligraphy* technique. Albers *et al.* [2] and Quesenbery [95] introduce five dimensions of usability (the 5Es) that are recommended in a usability evaluation. The 5Es are 'Effective', 'Efficient', 'Engaging', 'Error Tolerant' and 'Easy to Learn'. However, these dimensions are too generic and high-level to be used as guidelines for evaluating

complex 3D design and engineering applications [63, 61]. Thus, the following four criteria, which relate specifically to our cloud modelling system are used:

1. ease of use,
2. ease of learning,
3. modelling speed, and
4. realism of the generated result.

It is necessary for a number of subjects to evaluate the system, in order to achieve credible results. Nielsen [83] suggests that five users are able to find 85% of the usability problems and at least 15 users are required to discover all the problems. Therefore, we used 17 participants in our usability evaluation.
The first consideration in designing our user evaluation, was that we only have one input device (Wacom Intuos4 A4 tablet, that is reprogrammed using the Wacom API [110] so that it can be used in our 3D application). Our evaluation test could therefore only be administered to one subject at a time. Because of this limitation, our user evaluation was conducted using an observation method, using a similar procedure to that of Wei *et al.* [112], described in Section 7.5.
The second concern relates to the comparison of techniques. Our brush calligraphy method is a new modelling technique, which should ideally be compared with at least one existing modelling method under the same experimental circumstances. However, it would be too difficult and time consuming for users to learn an existing cloud modelling system in a short time. Most existing cloud modelling methods require users to remember and understand the detailed procedures. It would be more time than can be afforded in a meaningful user evaluation. Therefore, our evaluation focused only on the new modelling technique. In order to reduce the impact caused by a lack of comparisons, we made sure that:

1. all the participants must have a strong background in either human computer interaction or 3D modelling, and
2. we also provide users with video clips of other cloud modelling techniques before the experiment.

In this way, users are expected to have a clear understanding of the existing cloud modelling techniques.

7.2 Participant Recruitment

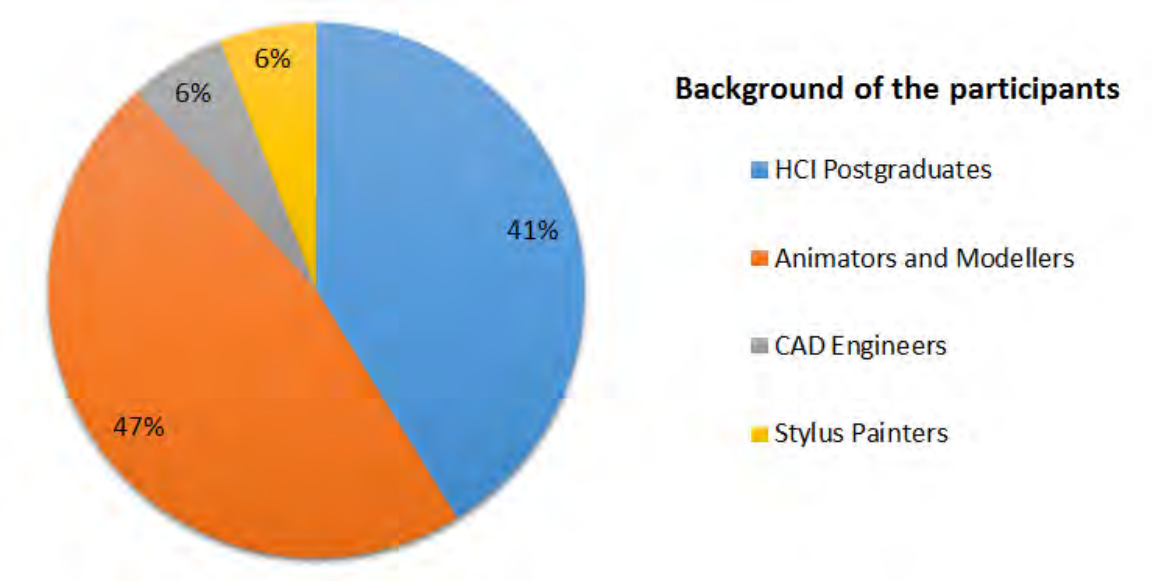


Figure 7.1: The total of 17 participants from both theoretical and practical backgrounds in 3D modelling systems.

Due to the demand for participants with a specific background, we first asked the computer science department for a recommendation list of ideal testing subjects. We also used a small amount of money from research funds so that we could offer the participants a minimum compensation (*R60*).

After receiving the names of potential participants, we sent each of them an email (Appendix D) to request their cooperation together with a brief introduction of our cloud modelling system. In order to provide participants with a better understanding of cloud modelling, we also included three video links of other cloud generation methods from existing modelling packages (Blender, 3D Max and Maya). More importantly, a time table was provided so that each participant could flexibly schedule their experiment session.

In the end, we were able to recruit 17 users for the user evaluation, as shown in Figure 7.1. These users can be divided into two groups:

1. **HCI postgraduates (7)** There were seven postgraduate students with an HCI background from the computer science department participating in our user evaluation. They are knowledgable in interaction design which should assist in finding both the explicit and implicit problems in using the painting style.
2. **Modellers and Artists (10)** There were two teachers and six senior students from a professional animation college, CityVarsity [17]. They are experienced in 3D modelling using Zbrush, Maya and Solidworks and had participated in many productions of commercial animations. Apart from this, we also invited a frequent stylus user in digital painting (Photoshop and Illustrator), and a postgraduate student from an engineering department that regularly uses AutoCAD.

7.3 Preliminary Preparations

Code Additions

Apart from the *Erase Mode* and miscellaneous functions, we also implemented two extra functions, particularly for data capture and UI improvements.

1. An export function was implemented so that user generated 3D clouds could be saved to a 'DDS' file for future reviews.
2. Apart from the camera control using the stylus, we implemented four shortcut-keys to alter the viewing direction immediately to the 'Top View', 'Front View', 'Left View' and 'Perspective View' by pressing the arrow keys.

Equipment Preparations

Due to the evaluation of subjects one at a time, an individual's behavior could be observed with great detail. Therefore, we prepared a digital DSLR camera, Nikon D3100, together with a tripod, so that operations of the stylus pen could be captured with high quality. Unfortunately, we overestimated the battery life of the camera.

After three experiments per day, we had to use a cellphone camera to continue the recording.

Pilot Test

A pilot test with an experienced programmer was conducted using the same procedures as the final user test to ensure the robustness of our system. One major error, which relates to the circle growth method was fixed before the user test. Through the pilot, it is determined that our system was ready for the final user experiment.

7.4 Environment Setup

The input device was a Wacom Intuos4 A4 tablet, which supports tilting and 1024 levels of pressure sensitivity. The test system was run on a laptop with i7-3610m (8GB RAM) and a GT 650M graphics card (2GB RAM).
Due to transportation problems, our experiments were held in two venues, computer science graduate laboratory of university (HCI students) and an office of the college (Animation College). In order to minimize the impact from the two different locations, both testing rooms were quiet and without interruptions. We noticed that most users were relaxed during the experimental sessions, perhaps as a result of the familiar environment.

7.5 Experimental Procedure

Each experiment was started by issuing each participant a consent form, requesting permission to use all the data collected from the experiment. All participants gave their consent. Each evaluation experiment took about 90 minutes and had four sections:

1. **Introduction**
   A brief introduction was given at the beginning of the session to explain what our system is about. One video clip of cloud generation using Blender was presented

so as to enhance their understanding of the cloud modelling background. This included further insight into comparative cloud modelling techniques.

2. **Guided Tutorial (Appendix A)**
   A 10-minute guided tutorial was provided, in which the users learnt to change canvas and camera and to generate clouds using the different painting styles. Participants followed a series of instructions to draw the Chinese character, Water. After that, they were provided with a free drawing stage in which questions were welcomed to help them familiarize with the system.
3. **Independent Modelling (Appendix B)**
   Users were then required to draw four clouds to match provided images, spending no more than 10 minutes on any one cloudscape. They were provided with a picture of a duck toy and three photos of real clouds.
   In this session, no questions were allowed and the subjects were required to model the cloud independently. The entire process, which involves pen use and cloud creation, was videotaped.
4. **Data Collection**
   There were four types of data collected in each session:
   - 'DDS' texture file that stores the 3D cloud.
   - 'AVI' video file that records the *Independent Modelling* section.
   - Questionnaire for statistical analysis.
   - Feedback in a written form.

7.6 Questionnaire Design (Appendix C)

Nielsen [83] suggests 10 guidelines for a standard heuristic evaluation. DeBoard [23] has designed a questionnaire based on these 10 guidelines and the '5Es' [95] to evaluate a user interface for online help, as shown in Figure 7.2.
These guidelines are well suited to general usability evaluation. However, the overall usability of our system is not our primary concern as the new brush calligraphy style is the main focus of evaluation. Therefore, our questionnaire is slightly modified to focus

1) Is it clear where on the screen to find the help system?

Very Unsatisfied _____1 _____2 _____3 _____4 _____5 Very Satisfied
Severity Ranking ____Low _____Moderate _____High Question is Not Applicable___

2) Is it clear how to begin to use the help system? (Is it obvious how to invoke the help?)

Very Unsatisfied _____1 _____2 _____3 _____4 _____5 Very Satisfied
Severity Ranking ____Low _____Moderate _____High Question is Not Applicable___

Figure 7.2: Questions designed by DeBoard [23] for a user interface evaluation.

specifically on brush calligraphy evaluation. It also uses questions with a Likert-type scale to reflect users' opinions on both the individual tasks and overall system. Users were required to provide their level of agreement (with strongly disagree:1, disagree:2, neutral:3, agree:4, strongly agree:5) to a question as shown in Figure 7.3.

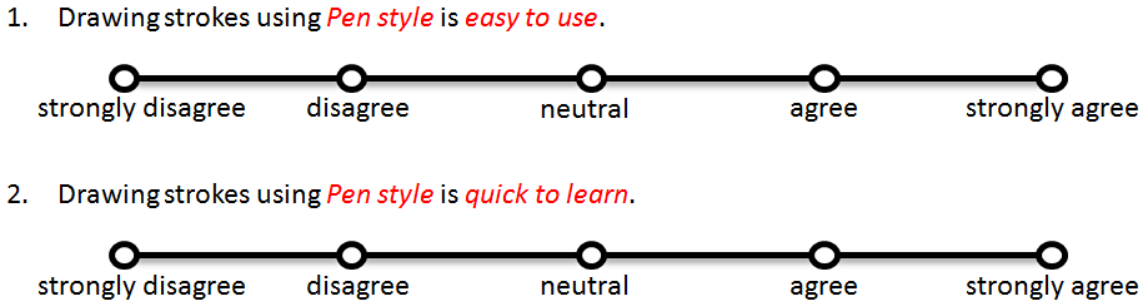


Figure 7.3: An example of the first two questions in the questionnaire, which is a Likert-type scale. These two questions aim to rate users' opinions on using the Pen style from the aspects of 'easy to use' and 'quick to learn'.

The questionnaire contains three groups. Each group focuses on various testing objects: brush calligraphy style, cloud drawing and system use. They are evaluated from the four aspects we extracted in the beginning of section 7.1. For example, the first group aims to evaluate the usability of our two painting styles: 2D Brush(Pen) and 3D Brush. The aspects of 'easy to use', 'quick to learn' and 'satisfactory results' are assessed for each painting style. The second group focuses on the cloud drawing section. Users are required to rate their agreement with efficient and effective cloud modelling for each

cloud. The last group evaluates the system use through the criteria of error tolerance, camera control, canvas adjustment, etc. These rating values from the questionnaire are collected for our statistical analysis.

7.7 Evaluation Results

The data collected from questionnaires, videos and saved 3D textures produce evaluation results in various ways. The questionnaires and video files enhance the statistical result which reflects user's satisfactory levels of each testing aspect. The 3D cloud dataset confirms their satisfaction particularly from a quality aspect. The written feedback reveals other problems of using this modelling system, which will be discussed at the end of this chapter.

7.7.1 Questionnaire Data

Table 7.1: Values from Likert-type questions Q1 to Q18(with strongly disagree:1, disagree:2, neutral:3, agree:4, strongly agree:5).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
1	5	5	4	5	5	4	4	4	4	4	4	4	3	4	4	3	4	4
2	5	5	4	5	4	4	4	5	4	3	3	3	5	4	5	2	3	3
3	2	4	2	4	3	4	4	4	4	4	3	4	5	5	3	2	1	2
4	4	4	4	4	4	4	4	3	3	3	3	3	3	4	4	3	2	3
5	4	4	4	4	4	4	3	4	4	4	3	4	4	2	4	5	2	5
6	3	5	5	4	4	4	4	4	4	3	4	4	3	3	3	2	3	3
7	4	4	4	5	5	4	5	5	4	4	4	4	3	3	4	3	3	4
8	4	4	4	4	4	4	4	5	4	4	3	4	4	4	4	4	3	4
9	5	5	5	5	5	5	4	4	5	5	4	4	5	4	4	5	5	5
10	4	4	3	4	4	4	5	5	4	4	4	4	4	4	4	3	4	4
11	5	4	4	4	5	4	2	2	3	5	5	4	5	5	5	5	5	4
12	4	5	4	5	5	5	5	5	5	5	4	4	5	4	5	3	4	4
13	5	5	5	3	4	2	5	5	5	4	4	4	4	5	4	2	2	2
14	4	5	5	4	5	5	5	4	4	3	4	4	4	5	4	1	2	4
15	2	4	1	4	3	3	5	4	5	3	3	3	4	4	3	2	2	3
16	4	5	4	5	5	5	4	5	4	4	4	5	5	5	5	5	4	5
17	3	4	4	4	4	4	4	4	4	2	2	2	3	3	3	4	4	4

Table 7.1 lists the data from the questions Q1 to Q18, as shown in Appendix C. These questions aim to reflect the user experience with the use of brush calligraphy style and cloud drawing.

Table 7.2: Values from Likert-type questions Q19 to Q27(with strongly disagree:1, disagree:2, neutral:3, agree:4, strongly agree:5).

	Q19	Q20	Q21	Q22	Q23	24	Q25	Q26	Q27
1	5	5	4	4	4	4	5	4	4
2	5	5	4	5	4	4	5	4	4
3	5	4	2	4	4	4	5	5	5
4	5	4	4	3	2	3	4	4	4
5	4	5	4	3	4	5	4	4	4
6	5	4	5	5	5	5	5	5	5
7	5	5	4	4	5	5	5	5	4
8	4	4	3	4	4	4	4	4	4
9	5	4	4	5	3	5	5	4	5
10	5	5	5	3	5	4	5	5	5
11	4	4	4	5	3	4	4	3	4
12	5	5	5	5	4	4	4	5	4
13	5	5	4	3	2	3	4	4	4
14	5	4	5	3	4	4	5	4	5
15	5	4	4	3	4	4	5	5	5
16	5	5	5	5	4	5	5	5	5
17	5	5	4	4	4	4	4	4	5

Table 7.2 lists the data from the questions Q19 to Q27, as shown in Appendix C. These questions aim to reflect the overall experience of using the prototype system.

7.7.2 Statistical Analysis

Table 7.3: Questionnaire results. Mean and standard deviation values from Likert-type questions (with strongly disagree:1, disagree:2, neutral:3, agree:4, strongly agree:5).

	Shape Realism	Visual Realism	Lag Acceptance	Easy to Use	Quick to Learn	User Friendly UI
AVG	4.2	4.4	3.8	4.4	4.4	4
STDDEV	0.84	0.55	0.84	0.55	0.89	0.7

Despite the short training time, most users were able to achieve competency by the time they had completed the first cloudscape. They also expressed a uniform preference for our 3D cloud modelling system over conventional modelling packages

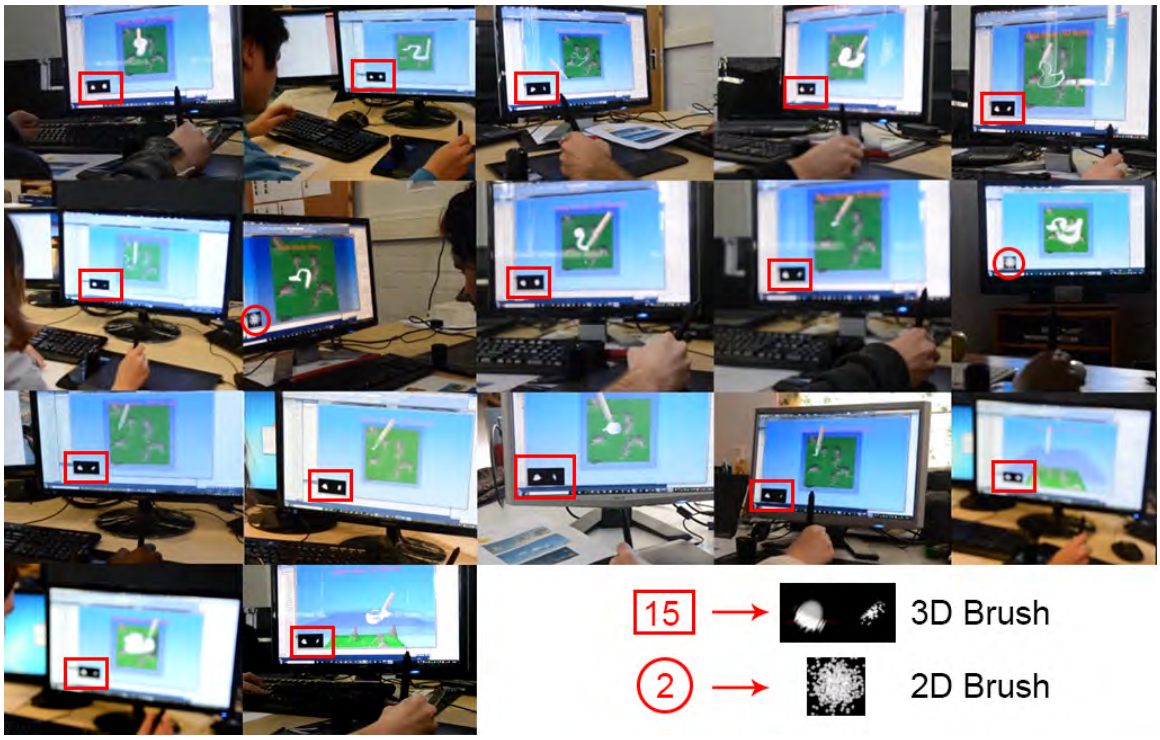


Figure 7.4: 17 users painting a duck-shaped cloud using our system during the experiment. It is observed that 15 users prefer the use of 3D brush in painting the duck cloud.

and rated the system consistently highly on realism, ease of use; and ease of learning (see Table 7.3). Our weakest result 3.8 on a scale of 1 to 5 related to the acceptance of the lag caused by stroke mapping, which was less than a second. This indicates that although currently acceptable, further acceleration of this process would be welcomed by users.

We also asked users to separately rate the 2D and 3D brushes on a scale from 1 to 5 with mean results of 3.88 for the 2D brush and 4.05 for the 3D Brush. A T-test with $p = 0.26$ indicates that there is no significant preference for using one of these two painting styles above the other. However, in practice we noticed that 15 out of 17 subjects used the 3D brush more than the 2D brush during their cloud drawing session. 90% of the clouds were painted through the use of the 3D brush by these 15 users. Figure 7.4 indicates this preference for using the 3D brush, which can be identified through the footprint sample texture at the bottom left of each screen.

The statistical results, as shown in Table 7.3, generally present a positive outcome from questionnaires and the video clips. In order to confirm these results, we loaded the user created 3D cloud dataset files into our modelling system. This allowed us to visualize the results for quality comparisons.

7.7.3 User Created 3D clouds

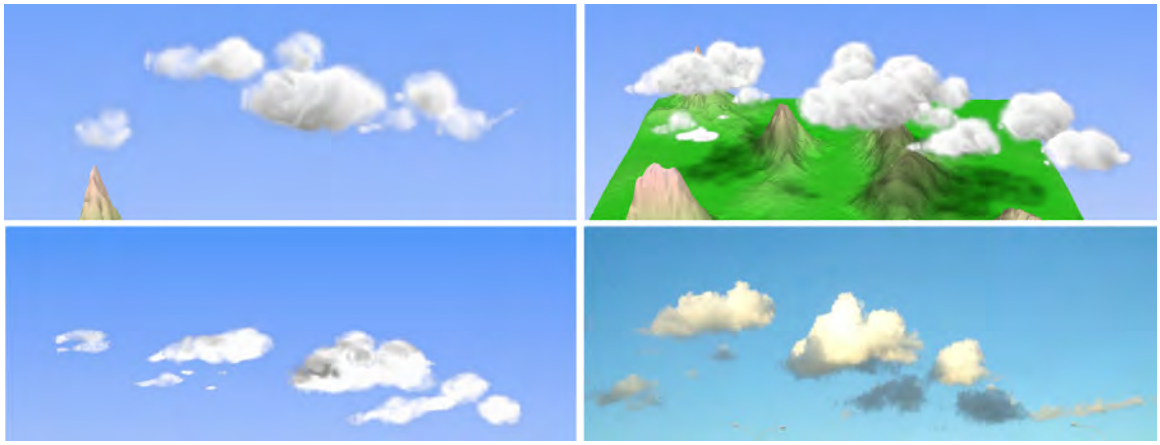


Figure 7.5: Cloudscapes painted under 10 minutes during user evaluation. The bottom right image is the original cloud from which the other three images were replicated.

Figure 7.5 shows several user drawn clouds based on the second photograph of natural clouds in the *Independent Modelling* section, see Appendix B. Note that the bottom right image is the natural photo. It was replicated by the users from various viewing angles. For instance, the top left cloudscape was created from the front view and the top right one is built from a slanted top-down direction. The bottom left image of Figure 7.5 was built from a top-down view. We observed that this user frequently switched the camera views between the front for viewing and top down for drawing. This cost him twice the time of the other modellers. Nevertheless, all these results were created in less than 10 minutes.

Users were required to rate Likert-type questions on a scale from 1 to 5, with mean and SD results of 4.06 ± 0.83 for 'easy to draw' and 4.0 ± 0.71 for 'satisfactory result'. This indicates most users were happy with using brush calligraphy for cloud generation. Although users had varied perceptions of the exact cloud locations and formations from

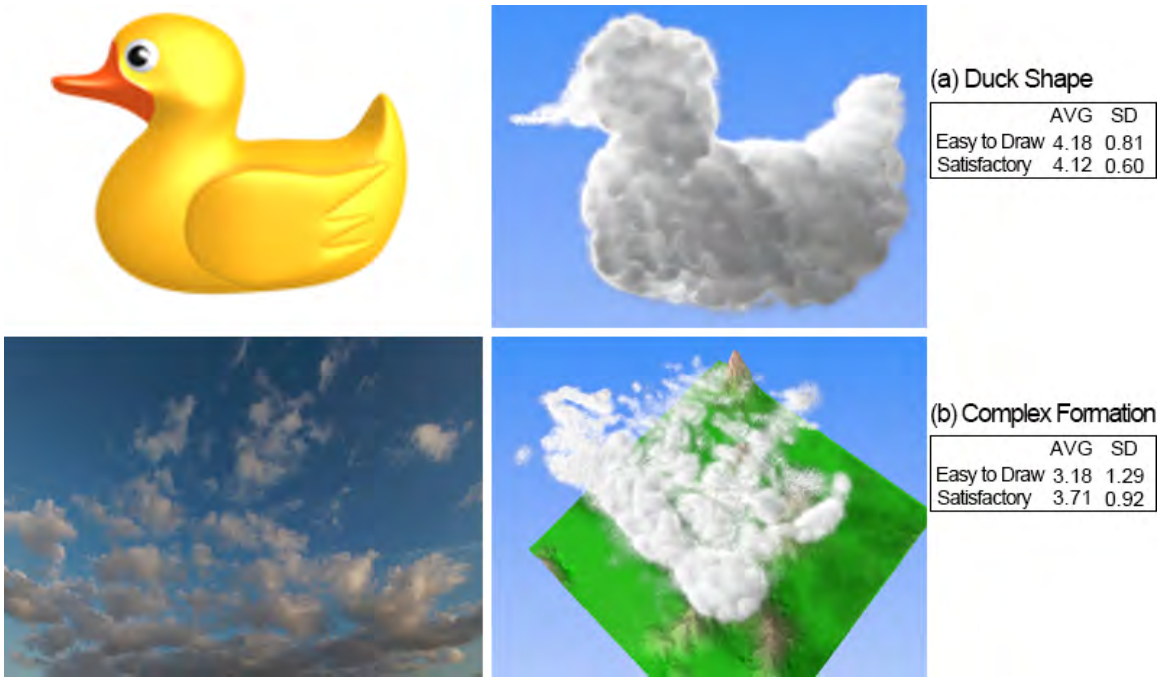


Figure 7.6: Other results with mean and SD values on a scale from 1 to 5: (a) duck shape with white 'rim' on the edge.(b) cloudscape with complex formation from natural photo 3.

the natural photo, the generated results approximately met with users' expectations, as most broad formation features were captured from the photo and replicated in 3D clouds. For instance, all the results contain a rounded cloud in the middle and small clouds with sharp tails on the edge. It is worth noting that none of the clouds in this photo were modelled with complete accuracy. Nevertheless, users were still largely satisfied with their work.

Figure 7.6 shows other user painted results from the *Independent Modelling* section with the list of related means and SD values. It is pleasing to notice that the duck image can be imitated with high fidelity. The questionnaire results also indicate the duck cloud was easy to draw and satisfactory to most modellers. The cloud B is a cloudscape with a complex formation, and was the last image in the experiment. However, modellers found this formation more difficult to paint compared to the previous samples. One of the issues could be that it is difficult to determine exact shape from a single photo. The results confirm this intricacy with an obvious decrease in both values.

7.7.4 Results Conclusion

In general, users are largely pleased by modelling 3D clouds using the brush calligraphy style. They started with simple stroke practice and gradually painted complex 3D clouds. After using this new modelling style for 90 minutes, their overall feedback on the four criteria were mainly positive:

1. *Ease of Use*

   Users are satisfied with using this method for drawing various types of clouds. It is different from other 3D modelling packages, which demand that users manually specify many properties such as the location and density of the cloud. Instead the brush calligraphy style only requires users to focus on the painting itself. There is no parameter tuning in this process, which is more intuitive and easy to use.
2. *Ease of Learning*

   Users find this modelling style easy to learn. During the experiment, all users were able to start generating their desired 3D clouds after 10 minutes of practice. We believe this largely benefits from the use of a traditional painting mechanism.
3. *Modelling Speed*

   All clouds in the experiment were finished within 10 minutes. Besides the intuitive painting mechanism for cloud generation, other editing functions such as Undo/Redo, Camera Auto Rotation and Erase also contribute to increasing the modelling speed. When drawing the duck shape cloud, two participants outlined the duck shape using only one stroke and completed this cloud modelling in less than one minute.
4. *Realism of the generated result*

   In the statistical results, there are two indictors for the realism of the generated 3D clouds. The first one 'Shape Realism' shows that the shape of the generated cloud is consistent with the user's expectation. The second indicator 'Visual Realism' shows that users are pleased with the final visual result of the generated cloud, which consists of the edge features and the light effects.

These positive results indicate that the four criteria are fulfilled for artists who model 3D clouds using the brush calligraphy style.

7.7.5 Problems from Feedback

Users generally produced positive feedback for the new modelling technique. However, three major problems were noted:

1. *Difficulty in drawing cirrostratus clouds*
   It is difficult for the user to create large scale cirrostratus clouds, as shown in Figure 7.7. In order to produce the cirrostratus using the new modelling technique, the user had to replicate a number of thin strokes so as to complete its formation little by little, which is a tedious task for most users.
2. *Difficulty in painting from the front view*
   When painting from a frontal view of the 3D canvas, the visible area of the canvas is limited due to projection transformations at that angle. This limits the space for cloud growth, which significantly increases the painting difficulty.
3. *Unintended cloud generation due to pen lift failure*
   The cloud mapping is triggered when the pen is lifted. This mapping is designed to execute after each stroke in order to achieve the best result. However, many users fail to lift the pen during their cloud painting as they were not accustomed to painting in this way. This leads to an unintended result.

These problems will need to be addressed in future research.

7.8 Summary

This chapter explains the implementation of our user evaluation experiment. In general, the experiment produced positive results indicating that users are satisfied with 3D cloud modelling using this painting style. They found the system easy to use and quick to learn. The generated 3D clouds had a realistic appearance that matched their expectations, even though a low-fidelity lighting model was used.

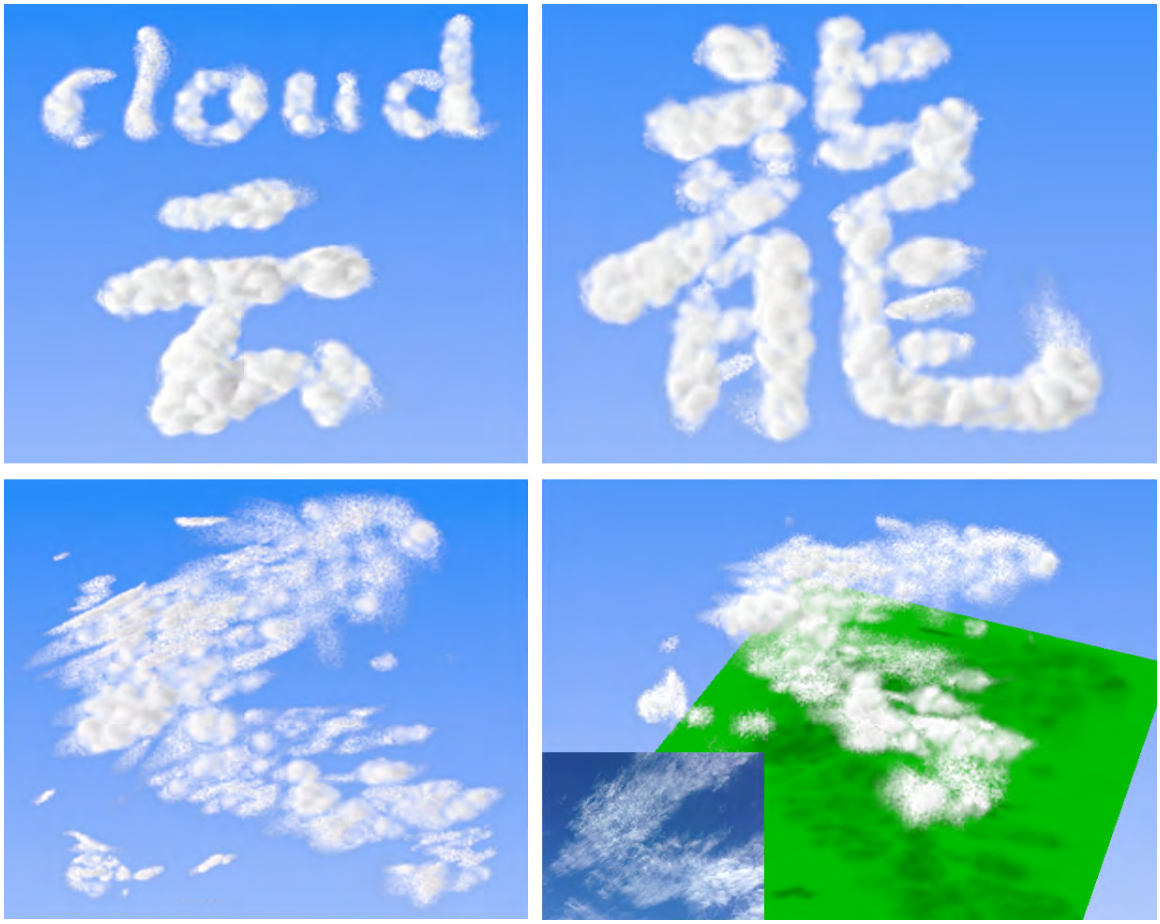
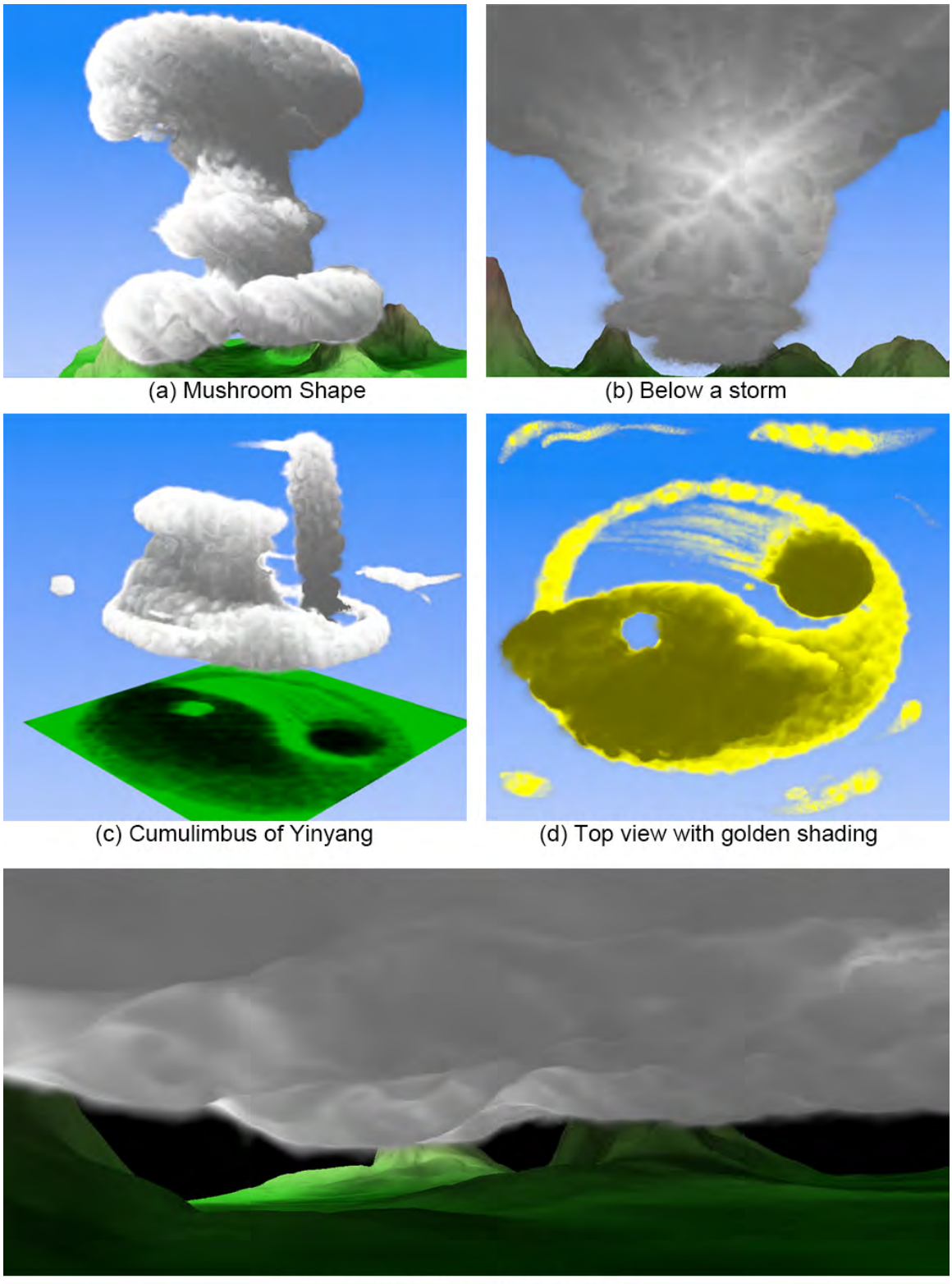


Figure 7.7: Character clouds and Cirrostratus creations without time limit.



(a) Mushroom Shape

(b) Below a storm

(c) Cumulimbus of Yinyang

(d) Top view with golden shading

(e) Front view below stratus clouds

Figure 7.8: Other artistic creations without time limit.

Chapter 8

Conclusions

This dissertation describes a volumetric cloud modelling system that uses a brush calligraphy style. This novel modelling technique is implemented by the integration of various modules, such as 3D cloud visualisation, brush simulation, a cloud growth method and miscellaneous other functions.
A number of novel approaches were proposed during the development of these modules. With respect to the five research questions posed in section 1.8, these new approaches were evaluated and the outcomes are reflected in the thesis statements below:

1. **Is the Chinese brush calligraphy style helpful in cloud generation?**
   Brushes possess unique features which are particularly helpful for cloud modelling.
   Inspired by calligraphy and Chinese landscape painting, we abstract elements of the Chinese brush to allow expressive and economical control over 3D cloud generation. The beauty of brush work derives from its intricate nature, which delivers features such as dynamic thickness along the stroke, furry effects along a bending edge, sharp hooks at the tails and a wispy footprint due to fast brush movement.
   In our cloud modelling system, these unique features are replicated as they provide effective surface detail for various types of clouds. This substantially improves the generation of clouds with satisfactory shape and formation features as determined by modellers.

2. **How can one simulate the Chinese brush at a low performance cost using a stylus pen?**

   Brush features are approximately simulated by using a stylus pen.

   The brush features derive from brush control through five mechanisms: the control of pressure, flow, speed, brush tilt and ink dispersion. At first we used a skeleton-based brush. Unfortunately, pilot user tests indicated severe confusion when using this brush due to inconsistent tip movement between the 3D brush and the stylus pen.

   We subsequently implemented a 2D and a 3D brush, based on texture sampling techniques. The 3D brush manipulates a complex volumetric footprint, which produces various brush features depending on the pen's pressure, speed and tilt. Therefore, these important brush features can be approximately simulated through the use of pen devices.

3. **How can one generate the desired volumetric clouds in a 3D scene based on the 2D footprint painted by the artists?**

   3D clouds are effectively generated from a 2D footprint.

   We propose a *Hybrid Mapping Method* that enables 3D cloud generation based on mapping from a painted 2D footprint. This method involves three stages: the growth of circles, the mapping of both circles and boundary pixels, and merging with a 3D noise texture.

   This *Hybrid Mapping Method* produces compelling volumetric clouds that were highly rated in user tests.

4. **Will the performance of volumetric rendering be fast enough for the support of cloud modelling in real-time?**

   It is feasible to perform both the volumetric data modification and realistic visualisation at interactive rates.

   We extend GPU-based volumetric rendering techniques [62] [18] with the support of an octree-based empty space skipping method and a fast volumetric illumination algorithm. This facilitates the rendering of 3D clouds with convincing quality in real-time.

   Furthermore, we amortize the computational cost of the cloud painting and

generation over both the CPU and GPU units in order to preserve interactive rates in cloud modelling.

5. **Is our cloud modelling system intuitive to use?**
   It is intuitive to model clouds using our brush calligraphy method.
   Our system supports the painting of different types of clouds by capturing features, such as shape, density and edge detail, from stroke footprints.
   The user evaluation experiment indicates that this *Brush Calligraphy* style can be quickly learnt and is easy to use. Users are able to intuitively and interactively create volumetric clouds in a similar way to landscape painting, in which artists are able to paint a cloud using only one brush, one colour and even in one stroke.

In conclusion, we have demonstrated a novel cloud modelling system using a *Brush Calligraphy* interface. The *Brush Calligraphy* style is effective for modelling 3D clouds. Our users found the style inspiring and performed extensive artistic cloud creations beyond those required for the user experiments. Therefore, we can conclude the original research question as:

A variety of visually compelling 3D clouds can be effectively and intuitively modelled by 3D artists using a Chinese brush calligraphy style.

To the best of our knowledge, this work is the first to implement a brush calligraphy method of 3D cloud generation. However, there remain a number of limitations which can be explored in future work.

8.1 Future Work

Application Extension

One possible application of future work could be the extension of this approach to other forms of procedural modelling, such as trees and terrain. In general, brush-based painting is well suited to creating objects with an amorphous volumetric structure, in particular those with unclear boundaries that are difficult to define by sketching.
In these systems, the voxel-based data structure may need to be converted to other

forms. This will require fundamental changes to both the core modules for visualisation and data generation. However, the use of the brush calligraphy style is potentially helpful for modelling trees and terrains on a large scale.

Ink Simulation

The 2D footprint in our brush simulation is directly captured from the shading of the bristle curves in the frame buffer. A high-fidelity ink simulation is unnecessary for our cloud generation, as explained in section 5.2.2.
However, the ink dispersion effect may play an important role in extended applications that adopt surface-based data structures. There are several current approaches in this area, such as the ink-wash painting style by Xu et al [119] and the use of scanned images by Kim et al. [60] and Lu et al. [73]. Although these methods provide interactive performance, a balanced implementation strategy is still necessary due to the extra costs of data generation and object rendering.

Cloud Animation

Our cloud modelling system is focused on static cloud generation, which is the ideal cloud state for the evaluation of the proposed *Brush Calligraphy* style.
However, our system uses a voxel-based data structure, which can be directly manipulated by several procedural methods such as cellular automata or computational fluid dynamics. This enables data changes such as the animation of cloud dispersion in real-time.
Alternatively, the procedural method can also be controlled in a manual style. In the Vacuum style of our Erase mode, we implemented a sphere-based cutting method to modify the dataset at the desired location. Instead of complete cutting, a more realistic vacuum style can be procedurally implemented by providing wispy features that reflect the wind impact. This can also enhance the simulation of wispy edges in cloud modelling.

Extended Skyline

Our cloud dataset is limited to 256^3 due to performance considerations. It is able to create a skyline of moderate size, sufficient for our research into a *Brush Calligraphy* style.

In future work, there are a number of methods that can be used to extend this size so as to leverage our brush method in large cloudscape creation. For instance, instead of using a single 3D texture to represent the entire cloud dataset, each painted 3D cloud with the stroke flow information can be stored in separate 3D textures. They can be rendered using various dimensions with regard to the viewing directions and distance. This could potentially not only provide better performance in rendering, but also enable an effectively infinite skyline.

Appendices

Appendix A: Guided Tutoring

1. Delete Data(Key: Delete)
2. Reset Camera(Key: R in Camera Mode)
3. Change to Canvas Mode (Key: 2)
4. Adjust the Canvas Altitude to 96.
5. Change to Camera Mode (Key:1)
6. Rotate to top-down painting view (Key: Up or Menu or Stylus Moving)
7. Change to Paint Mode (Key:3)
8. Change a Paint Style by pressing (Key: M or Pen Higher Button)
9. Paint a stroke on the left and release the pen.

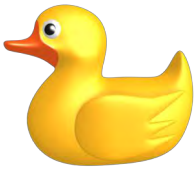
10. Check the Circle Growth Progress. (Key: C)
11. Switch the Circle Growth result by pressing Key: C again.
12. Paint a stroke on the right and release the pen.

13. Trigger the Undo to get rid of the right stroke: (Undo can be used by Key:Ctrl+Z)
14. Change to Vaccum Mode (Key:4)
15. Erase the Left Stroke
16. Change to Camera Mode (Key:1)
17. Reset Camera (Key: R)
18. Change to Canvas Mode (Key:2)
19. Adjust the Canvas Altitude to 64
20. Change to Camera Mode (Key:1)
21. Change Camera View To Top View (Key: Up)
22. Change to Paint Mode (Key:3)
23. Change a Paint Style by pressing (Key: M or Pen Higher Button)
24. Draw a calligraphy as close as you can(Remember to release the tip after each stroke)

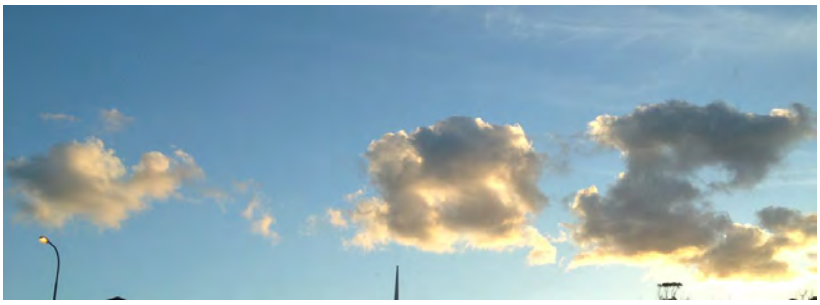
25. Change to Camera Mode to check you result (Key:1)
26. Save the data to your file by (invoke Menu by Stylus Pen's side higher button or keyboard: 'M', and select save to file)

Appendix B: Cloud Painting

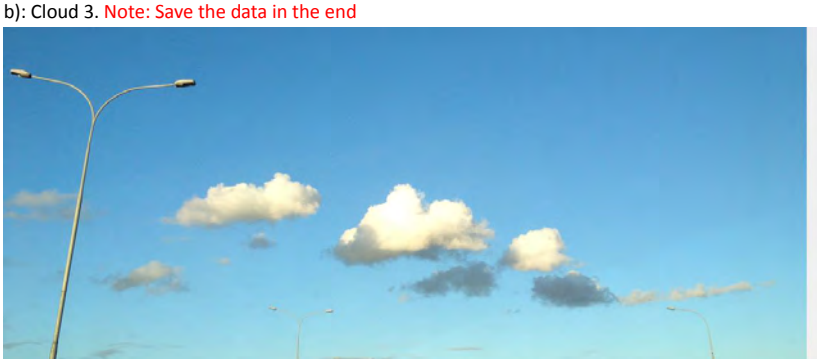
a): Cloud 1 . Note: Save the data in the end



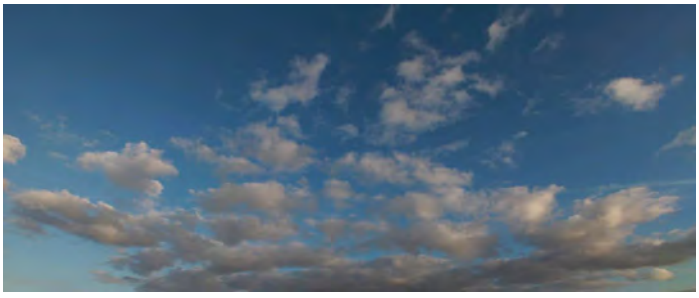
b): Cloud 2. Note: Save the data in the end



b): Cloud 3. Note: Save the data in the end



b): Cloud 4. Note: Save the data in the end



Appendix C: Questionnaire

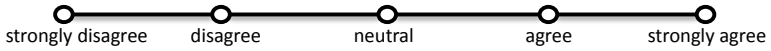
Age:

Gender:

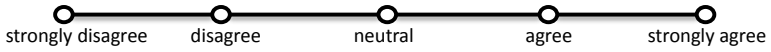
Your experience in 3D modelling:

Paint Style

1. Drawing strokes using *Pen style* is *easy to use*.



2. Drawing strokes using *Pen style* is *quick to learn*.



3. *Results* in using *Pen style* is *satisfactory*.

4. Drawing strokes using *3D Brush style* is *easy to use*.

5. Drawing strokes using *3D Brush style* is *quick to learn*.

6. *Results* in using *3D Brush style* is *satisfactory*.

Cloud Drawing

7. Cloud 1 is *easy to draw*.

8. Cloud 1 is *quick to model*.

9. Cloud 1 has a *satisfactory result*.

Appendix C: Questionnaire

10. Cloud 2 is *easy to draw*.

11. Cloud 2 is *quick to model*.

12. Cloud 2 has a *satisfactory result*.

13. Cloud 3 is *easy to draw*.

14. Cloud 3 is *quick to model*.

15. Cloud 3 has a *satisfactory result*.

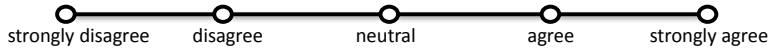
16. Cloud 4 is *easy to draw*.

17. Cloud 4 is *quick to model*.

18. Cloud 4 has a *satisfactory result*.

System Use

19. The *Undo/Redo* is *helpful*.



20. The *Camera View* is *easy to change*.

Appendix C: Questionnaire

21. The *Canvas* is *easy to change*.

22. The *Light effect* is *helpful*.

23. The *Erase mode* is *easy to use*.

24. The *Menu UI* is *easy to use*.

25. The *Menu* is *helpful*.

26. The *shape of generated 3D cloud* is *realistic*.

27. The *visual affect of generated 3D cloud* is *realistic*.

Appendix D: Email Request for User Evaluation

Hi Marina,

Thanks for your willingness in helping us with the evaluation experiments.
This experiment is about testing a new modelling method, which models 3D clouds using a Chinese Painting Style (Chinese brush simulation using stylus pen).
To help your understanding in cloud modelling, we reference 3 short video clips which involves cloud modelling using blender, 3D max and Maya

1. Blender: http://www.youtube.com/watch?v=EpGdRJFPVEI
2. 3D Max: http://www.youtube.com/watch?v=hy_oP6DRPiY
3. Maya: http://www.youtube.com/watch?v=NI0jAns2GZs

The experiment may take 80~90 mins. Here is the available slots for the experiments next week, please confirm your available time with me through the email: barrywei60@hotmail.com

24~27th Sep	Tuesday	Wednesday	Thursday	Friday
9:00	empty	empty		empty
11:00				empty

Many Thanks

PH.D.c Chen Wei
UNIVERSITY OF CAPE TOWN
Department of Computer Science

Bibliography

[1] Adobe. *Photoshop*. viewed 1 Februrary 2014, <http://www.photoshop.com>.

[2] Albers, M. and Mazur, B. *Content and Complexity: Information Design in Technical Communication*. Hillsdale, NJ, USA: Lawrence Erlbaum, 2003.

[3] Autodesk. *SoftImage*. viewed 1 Februrary 2014, <http://www.autodesk.com/products/autodesk-softimage>.

[4] Batcho, P. and Schlick, T. "Special stability advantages of position-Verlet over velocity-Verlet in multiple-time step integration". In: *Journal of Chemical Physics* 115.9 (Sept. 2001), pp. 4019–4029.

[5] Baxter, B., Scheib, V., Lin, M. C., and Manocha, D. "dab: interactive haptic painting with 3d virtual brushes". In: *Proceedings of the 28th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 461–468.

[6] Baxter, W. and Govindaraju, N. "simple data-driven modeling of brushes". In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, DC, USA: ACM, 2010, pp. 135–142.

[7] Baxter, W., Wendt, J., and Lin, M. C. "impasto: a realistic, interactive model for paint". In: *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*. NPAR '04. Annecy, France: ACM, 2004, pp. 45–148.

[8] Blake, R. and Sekuler, R. *Perception*. McGraw-Hill higher education. McGraw-Hill Companies,Incorporated, 2006.

[9] Blender. *Blender*. viewed 1 Februrary 2014, <http://www.blender.org>.

[10] Bouthors, A. and Neyret, F. "modeling clouds shape". In: *Eurographics (short papers)*. Ed. by M. Alexa, E. G. 2004.

[11] Bouthors, A., Neyret, F., Max, N., Bruneton, E., and Crassin, C. "interactive multiple anisotropic scattering in clouds". In: *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*. I3D '08. Redwood City, California: ACM, 2008, pp. 173–182.

[12] Cabral, B., Cam, N., and Foran, J. "accelerated volume rendering and tomographic reconstruction using texture mapping hardware". In: *Proceedings of the 1994 Symposium on Volume Visualization*. VVS '94. Tysons Corner, Virginia, USA: ACM, 1994, pp. 91–98.

[13] Chu, N. and Tai, C.-L. "an efficient brush model for physically-based 3d painting". In: *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*. PG '02. Washington, DC, USA: IEEE Computer Society, 2002, p. 413.

[14] Chu, N. and Tai, C.-L. "Real-Time Painting with an Expressive Virtual Chinese Brush". In: *IEEE Computer Graphics Applications* 24.5 (Sept. 2004), pp. 76–85.

[15] Chu, N. and Tai, C.-L. "moxi: real-time ink dispersion in absorbent paper". In: *Proceedings of the 32th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '05. Los Angeles, California: ACM, 2005, pp. 504–511.

[16] Chu, N., Baxter, W., Wei, L.-Y., and Govindaraju, N. "detail-preserving paint modeling for 3d brushes". In: *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*. NPAR '10. Annecy, France: ACM, 2010, pp. 27–34.

[17] CityVarsity. *Cityvarsity: school of media and creative arts*. viewed 1 Februrary 2014, <http://www.cityvarsity.co.za/homepage.php>.

[18] Crane, K., Llamas, I., and Tariq, S. "Chapter 30: Real-Time Simulation and Rendering of 3D Fluids". In: Nguyen, H. *GPU Gems 3*. Addison-Wesley, 2007.

[19] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. "gigavoxels: ray-guided streaming for efficient and detailed voxel rendering". In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. Boston, MA, USA: ACM, 2009, pp. 15–22.

[20] Crawfis, R. and Max, N. "texture splats for 3d scalar and vector field visualization". In: *Proceedings of the IEEE Conference on Visualization '93*. VIS '93. San Josa, CA, USA, 1993, pp. 261–266.

[21] Dan, M. and Ning, K. *Culture Records of Central Plains*. Shanghai People's Publishing House, 1998, p. 306.

[22] Das, K., Diaz-Gutierrez, P., and Gopi, M. "sketching free-form surfaces using network of curves". In: *Proceedings of the 2nd Eurographics Workshop on Sketch-based Interfaces and Modeling*. SBIM '05. Trinity College Dublin, Ireland: ACM, 2005, pp. 127–134.

[23] DeBoard, D. "Heuristic to Evaluate Online Help". In: *Usability Interface* 10.3 (Jan. 2004), pp. 18–27.

[24] DiVerdi, S., Krishnaswamy, A., Mech, R., and Ito, D. "painting with polygons: a procedural watercolor engine". In: *IEEE Transactions on Visualization and Computer Graphics* 19.5 (May 2013), pp. 723–735.

[25] DiVerdi, S., Krishnaswamy, A., Mech, R., and Ito, D. "a lightweight, procedural, vector watercolor painting engine". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. Costa Mesa, CA, USA: ACM, 2012, pp. 63–70.

[26] Dobashi, Y., Kaneda, K., Yamashita, H., Okita, T., and Nishita, T. "a simple, efficient method for realistic animation of clouds." In: *Proceedings of the 27th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New Orleans, LA, USA: ACM, 2000, pp. 19–28.

[27] Dobashi, Y., Kusumoto, K., Nishita, T., and Yamamoto, T. "feedback control of cumuliform cloud formation based on computational fluid dynamics". In: *Proceedings of the 35th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '08. Los Angeles, CA, USA: ACM, 2008, 94:1–94:8.

[28] Dobashi, Y., Shinzo, Y., and Yamamoto, T. "Modeling of Clouds from a Single Photograph." In: *Computer Graphics Forum* 29.7 (Oct. 2010), pp. 2083–2090.

[29] Dobashi, Y., Iwasaki, W., Ono, A., Yamamoto, T., Yue, Y., and Nishita, T. "An Inverse Problem Approach for Automatically Adjusting the Parameters for Rendering Clouds Using Photographs". In: *ACM Transactions on Graphics* 31.6 (Nov. 2012), 145:1–145:10.

[30] Dorsey, J., Xu, S., Smedresman, G., Rushmeier, H. E., and McMillan, L. "the mental canvas: a tool for conceptual architectural design and analysis." In: *Pacific Conference on Computer Graphics and Applications*. PG '07. Maui, HI, USA: IEEE Computer Society, 2007, pp. 201–210.

[31] Ebert, D. S. "volumetric modeling with implicit functions: a cloud is born". In: *ACM SIGGRAPH 97 Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH '97*. SIGGRAPH '97. Los Angeles, CA, USA: ACM, 1997, p. 147.

[32] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. *Texturing and Modeling: A Procedural Approach*. 3rd edition. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[33] Elek, O., Ritschel, T., Wilkie, A., and Seidel, H.-P. "interactive cloud rendering using temporally-coherent photon mapping". In: *Proceedings of Graphics Interface 2012*. GI '12. Toronto, Ontario, Canada: Canadian Information Processing Society, 2012, pp. 141–148.

[34] Elinas, P. and Stürzlinger, W. "Real Time Rendering of 3D Clouds". In: *The Journal of Graphics Tools* 5.4 (Nov. 2000), pp. 33–45.

[35] Engel, K., Kraus, M., and Ertl, T. "high-quality pre-integrated volume rendering using hardware-accelerated pixel shading". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '01. Los Angeles, California, USA: ACM, 2001, pp. 9–16.

[36] Engel, K., Hadwiger, M., Kniss, J. M., Salama, C. R., and Weiskopf, D. *Real-time Volume Graphics*. 1st edition. Natick, MA, USA: A. K. Peters, Ltd., 2006.

[37] Ernst, M., Vogelgsang, C., and Greiner, G. "stack implementation on programmable graphics hardware". In: *Proceedings of Vision Modeling and Visualiztion Conference 2004*. VMV '04. Stanford, CA, USA: Aka GmbH, 2004, pp. 255–262.

[38] Farrer, A. *The Brush dances & the ink sings: Chinese paintings and calligraphy from the British Museum*. Southbank Centre, London, UK: Haywood Gallery, 1990.

[39] Foley, T. and Sugerman, J. "kd-tree acceleration structures for a gpu raytracer". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. Los Angeles, CA, USA: ACM, 2005, pp. 15–22.

[40] Gailly, J. and Adler, M. *A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. viewed 1 Februrary 2014, <http://www.zlib.net>.

[41] Gardner, G. Y. "visual simulation of clouds". In: *Proceedings of the 12th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '85. New York, NY, USA: ACM, 1985, pp. 297–304.

[42] Gobbetti, E., Marton, F., Antonio, J., and Guitián, I. "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets". In: *Visual Computer* 24.7 (July 2008), pp. 797–806.

[43] Green, S. *Volumetric Particle Shadows*. Tech. rep. nVidia, 2008.

[44] Guan, J. "'Gun Xue' Should Be Written by Cao Cao". In: *Journal of Beijing Institute of Education* 16 (Jan. 2002), pp. 63–70.

[45] Harris, M., Sengupta, S., and Owens, J. "Chapter 39: Parallel prefix sum (scan) with CUDA". In: Nguyen, H. *GPU Gems 3*. Addison-Wesley, 2007, pp. 851–876.

[46] Harris, M. *Optimizing Parallel Reduction in CUDA*. Tech. rep. nVidia, 2008.

[47] Harris, M. J. and Lastra, A. "Real-Time Cloud Rendering". In: *Computer Graphics Forum* 20.3 (Sept. 2001), pp. 76–84.

[48] Harris, M. J. "Real-time cloud simulation and rendering". The University of North Carolina at Chapel Hill. PhD thesis. 2003.

[49] Hoffman, D. D. *Visual Intelligence: How We Create What We See*. New York, NY, USA: W. W. Norton & Company, 2000.

[50] Horn, D. "Chapter 36. Stream reduction operations for GPGPU applications". In: Pharr, M. and Fernando, R. *GPU Gems 2*. Addison-Wesley, 2005, pp. 573–589.

[51] Horst, J. A. and Beichl, I. "a simple algorithm for efficient piecewise linear approximation of space curves." In: *Proceedings of the 1997 International Conference on Image Processing*. ICIP '97. Santa Barbara, CA, USA: IEEE, 1997, pp. 744–747.

[52] Hu, L. "Discuss the isostructure of Chinese calligraphy and landscape painting". MA thesis. Harbin Normal University, 2010.

[53] Huang, J., Crawfis, R., Shareef, N., and Mueller, K. "fastsplats: optimized splatting on rectilinear grids". In: *Proceedings of the 2000 IEEE Conference on Visualization*. VIS '00. Salt Lake City, UT, USA: IEEE Computer Society Press, 2000, pp. 219–226.

[54] Huang, S. wen, Way, D. lor, and Shih, Z. chung. "physical-based model of ink diffusion in chinese ink paintings". In: *Proceedings of the 11th International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*. WSCG '03. University of West Bohemia, Pilsen, Czech Republic: Journal of WSCG, 2003.

[55] Huerta, R. *Vermeer and Plato: Painting the Ideal*. Lewisburg, PA, USA: Bucknell University Press, 2005.

[56] Igarashi, T., Matsuoka, S., and Tanaka, H. "teddy: a sketching interface for 3d freeform design". In: *Proceedings of the 26th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. Los Angeles, CA, USA: ACM, 1999, pp. 409–416.

[57] Ikits, M., Kniss, J., Lefohn, A., and Hansen, C. "Chapter 39: Volume Rendering Techniques". In: Fernando, R. *GPU Gems*. Addison-Wesley, 2004, pp. 667–692.

[58] Jansen, J. and Bavoil, L. "fourier opacity mapping". In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, DC, USA: ACM, 2010, pp. 165–172.

[59] Kaneko, K. *Theory and applications of coupled map lattices*. 1st edition. Nonlinear Science. John Wiley & Sons, 1993.

[60] Kim, M. and Shin, H. J. "An Example-based Approach to Synthesize Artistic Strokes using Graphs". In: *Computer Graphics Forum* 29.7 (Sept. 2010), pp. 2145–2152.

[61] Kosmadoudi, Z., Lim, T., Ritchie, J., Louchart, S., Liu, Y., and Sung, R. "Review: Engineering Design Using Game-enhanced CAD: The Potential to Augment the User Experience with Game Elements". In: *Computer Aided Design* 45.3 (Mar. 2013), pp. 777–795.

[62] Kruger, J. and Westermann, R. "acceleration techniques for gpu-based volume rendering". In: *Proceedings of the 14th IEEE Visualization*. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 287–292.

[63] Lee, G., Eastman, C. M., Taunk, T., and Ho, C.-H. "Usability Principles and Best Practices for the User Interface Design of Complex 3D Architectural Design and Engineering Tools". In: *International Journal of Human-Computer Studies* 68.1-2 (Jan. 2010), pp. 90–104.

[64] Lee, J. "Physically-based modeling of brush painting". In: *Computer Networks and ISDN Systems* 29.14 (Oct. 1997), pp. 1571 –1576.

[65] Lee, J. "Diffusion rendering of black ink paintings using new paper and ink models." In: *Computers and Graphics* 25.2 (Apr. 2001), pp. 295–308.

[66] Lewis, J. P. "algorithms for solid noise synthesis". In: *Proceedings of the 16th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '89. New York, NY, USA: ACM, 1989, pp. 263–270.

[67] Li, W., Mueller, K., and Kaufman, A. "empty space skipping and occlusion clipping for texture-based volume rendering". In: *Proceedings of the 14th IEEE Visualization 2003*. VIS '03. Seattle, WA, USA: IEEE Computer Society, 2003, pp. 317–324.

[68] Li, X. "Discuss Cao Cao's calligraphy and artistic conception from the analysis and the appreciation of 'Gun Xue'". In: *Journal of Xuchang Vocational Technical College* 7.3–4 (Dec. 2011), pp. 71–78.

[69] Li, X. "Rendering Technology of 3D Digital Chinese Ink-Wash Landscape Paintings Based on Maya." In: *Transactions on Edutainment IX*. Ed. by Pan, Z., Cheok, A., MÃijller, W., and Liarokapis, F. Vol. 7544. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 150–159.

[70] Liao, H.-S., Chuang, J.-H., and Lin, C.-C. "efficient rendering of dynamic clouds". In: *Proceedings of the 2004 ACM SIGGRAPH International Conference on Virtual Reality Continuum and Its Applications in Industry*. VRCAI '04. Singapore: ACM, 2004, pp. 19–25.

[71] Lin, H. and Wang, C. *Skills of cloud in landscape painting*. Tianjing People's Fine Arts Publishing House, 2004.

[72] Liu, Z., Wang, Z., and Zhang, C. "scheme of dynamic clouds generation for 3d real time flight simulation". In: *Proceedings of the 2nd International Conference on Computer Modeling and Simulation 2010*. Vol. 2. ICCMS '10. Sanya, Hainan, China: IEEE Computer Society, 2010, pp. 370–374.

[73] Lu, J., Barnes, C., DiVerdi, S., and Finkelstein, A. "RealBrush: Painting with Examples of Physical Media". In: *ACM Transactions on Graphics* 32.4 (July 2013), 117:1–117:12.

[74] Lu, T. K. and Huang, Z. "a gpu-based method for real-time simulation of eastern painting". In: *Proceedings of the 5th International Conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. GRAPHITE '07. Perth, Australia: ACM, 2007, pp. 111–118.

[75] Lungu, I., Dana-Mihaela, P., and Alexandru, P. "Optimization Solutions for Improving the Performance of the Parallel Reduction Algorithm Using Graphics Processing Units". In: *Informatica Economica* 16.3 (Sept. 2012), pp. 72–86.

[76] Meißner, M., Hoffmann, U., and Straßer, W. "enabling classification and shading for 3d texture mapping based volume rendering using opengl and extensions". In: *Proceedings of the Conference on Visualization '99*. VIS '99. San Francisco, CA, USA: IEEE Computer Society Press, 1999, pp. 207–214.

[77] Microsoft. *Microsoft Flight Simulator 2004*. viewed 1 Februrary 2014, <http://www.microsoft.com/games/flight>.

[78] Microsoft. *Microsoft Fresh Paint*. viewed 1 Februrary 2014, <http://www.microsoft.com/en-us/freshpaint/default.html>.

[79] Miyazaki, R., Yoshida, S., Nishita, T., and Dobashi, Y. "a method for modeling clouds based on atmospheric fluid dynamics". In: *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*. PG '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 363–372.

[80] Morton, G. M. "a computer oriented geodetic data base and a new technique in file sequencing". In: *IBM Germany Scientific Symposium Series*. 1966.

[81] Mueller, K., Möller, T., and Crawfis, R. "splatting without the blur". In: *Proceedings of the IEEE Conference on Visualization '99*. VIS '99. San Francisco, CA, USA: IEEE Computer Society Press, 1999, pp. 363–370.

[82] Nagel, K. and Raschke, E. "Self-organizing criticality in cloud formation?" In: *Physica A: Statistical Mechanics and its Applications* 182.4 (Apr. 1992), pp. 519–531.

[83] Nielsen, J. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[84] Nishita, T., Dobashi, Y., and Nakamae, E. "display of clouds taking into account multiple anisotropic scattering and sky light". In: *Proceedings of the 23rd International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 379–386.

[85] Okabe, Y., Saito, S., and Nakajima, M. "paintbrush rendering of lines using hmms". In: *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. GRAPHITE '05. Dunedin, New Zealand: ACM, 2005, pp. 91–98.

[86] Olsen, L., Samavati, F. F., Sousa, M. C., and Jorge, J. A. "Technical Section: Sketch-based Modeling: A Survey". In: *Computer Graphics* 33.1 (Feb. 2009), pp. 85–103.

[87] Omelyan, I. P., Mryglod, I. M., and Folk, R. "Optimized Verlet-like algorithms for molecular dynamics simulations". In: *Physical Review E* 65.5 (May 2002), 056706:1–056706:5.

[88] OpenMP. *OpenMP*. viewed 1 Februrary 2014, <http://http://openmp.org/wp>.

[89] Orbay, G. and Kara, L. B. "sketch-based modeling of smooth surfaces using adaptive curve networks". In: *Proceedings of the 8th Eurographics Symposium on Sketch-Based Interfaces and Modeling*. SBIM '11. Vancouver, British Columbia, Canada: ACM, 2011, pp. 71–78.

[90] Overby, D., Melek, Z., and Keyser, J. "interactive physically-based cloud simulation". In: *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*. PG '02. Beijing, China: IEEE Computer Society, 2002, pp. 469–470.

[91] Perlin, K. "an image synthesizer". In: *Proceedings of the 12th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '85. New York, NY, USA: ACM, 1985, pp. 287–296.

[92] Perlin, K. "improving noise". In: *Proceedings of the 29th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, TX, USA: ACM, 2002, pp. 681–682.

[93] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing". In: *Computer Graphics Forum* 26.3 (Sept. 2007), pp. 415–424.

[94] Prasad, L. "Morphological Analysis of Shapes". In: *CNLS Newsletter* 139 (July 1997), pp. 1–18.

[95] Quesenbery, W. *Using the 5Es to Understand Users*. viewed 1 Februrary 2014, <http://www.wqusability.com/articles/getting-started.html>.

[96] Rawson, J. *The British Museum book of Chinese art*. 2nd edition. British Museum Press, 2007.

[97] Reeves, W. T. "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects". In: *ACM Transactions on Graphics* 2.2 (Apr. 1983), pp. 359–376.

[98] Reeves, W. T. and Blau, R. "approximate and probabilistic algorithms for shading and rendering structured particle systems". In: *Proceedings of the 12th Inernational Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '85. New York, NY, USA: ACM, 1985, pp. 313–322.

[99] Roditakis, A. "modeling and visualization of clouds from real world data." In: *Proceedings of the 20th International Congress for Photogrammetry and Remote sensing*. ISPRS '04. Istanbul, Turkey, 2004, pp. 658–663.

[100] Rubin, L., Duncan, J., and Herbert, H. *The Weather Wizard's Cloud Book: How You Can Forecast the Weather Accurately and Easily by Reading the Clouds*. Algonquin Books of Chapel Hill, 1989.

[101] Ruijters, D. and Vilanova, A. "optimizing gpu volume rendering". In: *Inproceedings of the 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2006*. WSCG '06. Plzen, Czech Repulic: UNION Agency, 2006.

[102] Schaufler, G. "dynamically generated impostors". In: *Proceedings of the GI Workshop "Modeling–Virtual Worlds–Distributed Graphics"*. MVD '95. Sankt Augustin, German: Infix-Verlag, 1995, pp. 129–135.

[103] Schpok, J., Simons, J., Ebert, D. S., and Hansen, C. "a real-time cloud modeling, rendering, and animation system". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '03. San Diego, CA, USA: Eurographics Association, 2003, pp. 160–166.

[104] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. "scan primitives for gpu computing". In: *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. GH '07. San Diego, CA, USA: Eurographics Association, 2007, pp. 97–106.

[105] Simhon, S. and Dudek, G. "sketch interpretation and refinement using statistical models". In: *Proceedings of the 15th Eurographics Conference on Rendering Techniques*. EGSR '04. Norrköping, Sweden: Eurographics Association, 2004, pp. 23–32.

[106] Stiver, M., Baker, A., Runions, A., and Samavati, F. "sketch based volumetric clouds". In: *Proceedings of the 10th International Conference on Smart Graphics*. SG '10. Banff, Canada: Springer-Verlag, 2010, pp. 1–12.

[107] Stocco, L. J. and Schrack, G. "integer dilation and contraction for quadtrees and octrees". In: *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*. PACRIM '95. Victoria, BC, Canada: IEEE, 1995, pp. 426–428.

[108] Stocco, L. J. and Schrack, G. "On Spatial Orders and Location Codes". In: *IEEE Transactions on Computers* 58.3 (Mar. 2009), pp. 424–432.

[109] Strassmann, S. "hairy brushes". In: *Proceedings of the 13th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 225–232.

[110] Wacom. *Wacom Developer Relations*. viewed 1 Februrary 2014, <http://http://us.wacom.com/developerrelations>.

[111] Wang, N. "realistic and fast cloud rendering in computer games". In: *Proceedings of the 30th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '03. San Diego, CA, USA: ACM, 2003, pp. 21–40.

[112] Wei, C., Marsden, G., and Gain, J. "novel interface for first person shooting games on pdas". In: *Proceedings of the 20th Australasian Conference on Computer-Human Interaction: Designing for Habitus and Habitat*. OZCHI '08. Cairns, Australia: ACM, 2008, pp. 113–121.

[113] Westermann, R. and Ertl, T. "efficiently using graphics hardware in volume rendering applications". In: *Proceedings of the 25th International Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. Orlando, FL, USA: ACM, 1998, pp. 169–177.

[114] Westover, L. "footprint evaluation for volume rendering". In: *Proceedings of the 17th Inernational Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '90. Dallas, TX, USA: ACM, 1990, pp. 367–376.

[115] Wither, J., Bouthors, A., and Cani, M.-P. "rapid sketch modeling of clouds". In: *Proceedings of the Fifth Eurographics conference on Sketch-Based Interfaces and Modeling*. SBM '08. Annecy, France: Eurographics Association, 2008, pp. 113–118.

[116] WorldMeteorologicalOrganization. *International Cloud Atlas*. 1st edition. International Cloud Atlas. Geneva: World Meteorological Organization, 1956.

[117] Xie, N., Laga, H., Saito, S., and Nakajima, M. "ir2s: interactive real photo to sumi-e". In: *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*. NPAR '10. Annecy, France: ACM, 2010, pp. 63–71.

[118] Xu, S., Tang, M., Lau, F. C. M., and Pan, Y. "virtual hairy brush for painterly rendering". In: *Graphical Models* 66.5 (Sept. 2004), pp. 263–302.

[119] Xu, T.-C., Yang, L.-J., and Wu, E.-H. "stroke-based real-time ink wash painting style rendering for geometric models". In: *Proceedings of SIGGRAPH Asia 2012 Technical Briefs*. SA '12. Singapore, Singapore: ACM, 2012, 19:1–19:4.

[120] Yeh, J.-s., Lien, T.-y., and Ouhyoung, M. "on the effects of haptic display in brush and ink simulation for chinese painting and calligraphy". In: *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*. PG '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 439–441.

[121] Young, E. *DirectCompute Optimizations and Best Practices*. Tech. rep. nVidia, 2010.

[122] Zhang, Y. and Ma, K.-L. "fast global illumination for interactive volume visualization". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '13. Orlando, FL, USA: ACM, 2013, pp. 55–62.

[123] Zhou, K., Ren, Z., Lin, S., Bao, H., Guo, B., and Shum, H.-Y. "real-time smoke rendering using compensated ray marching". In: *Proceedings of the 35th International Conference and Exhibition on Computer Graphics and Interactive Techniques*. SIGGRAPH '08. Los Angeles, CA, USA: ACM, 2008, 36:1–36:12.