


5-2014

Volumetric Cloud Rendering: An Animation of Clouds

Zhaoxin Ye

Clemson University, zhaoxin@g.clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

 Part of the [Communication Commons](#), [Computer Sciences Commons](#), and the [Film and Media Studies Commons](#)

Recommended Citation

Ye, Zhaoxin, "Volumetric Cloud Rendering: An Animation of Clouds" (2014). *All Theses*. 1924.
https://tigerprints.clemson.edu/all_theses/1924

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

VOLUMETRIC CLOUD RENDERING: AN ANIMATION OF CLOUDS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

by
Zhaoxin Ye
May 2014

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Donald House
Dr. Timothy Davis

Abstract

This paper demonstrates a production workflow for a volumetric-rendering-based short animation about clouds. The animation is based on the concept of a giant fish swimming in the sky from Zhuangzi's philosophical story. The algorithm and implementation for the modeling and rendering of clouds are also presented. A renderer was developed that uses the OpenVDB library for data storage, fast retrieving and grid manipulation. A user-friendly pipeline was also developed for cloud modeling and rendering, which used Python and XML for adjusting rendering parameters.

The pipeline includes Maya to build the rough cloud model and Houdini to calculate the interior light points. Final compositing was done in Nuke. Several MEL and Python scripts were also used to retrieve camera and light information from Maya and Houdini, thereby facilitating the production process.

Acknowledgments

I would like to acknowledge Dr. Jerry Tessendorf who has been giving me valuable guidance in volumetric rendering. His Physically Based Effects class provided me with the essential knowledge of how to build a volumetric renderer. I also used and modified his C++ classes like Camera, Volume, Color, PerlinNoise, Image, OIIOFiles, Utilities and CmdLineFind class in my renderer. My implementation for doing cloud rendering is based on his method for making clouds in feature film production. I also thank him for his wise and critical suggestions from each week's thesis meeting.

I thank DreamWorks Animation Studio whose engineers developed and maintained OpenVDB. I want to thank Jeff Budsberg for his presentation during the DreamWorks/Digital Production Arts Summer 2013 course on the Clemson University campus about the clouds in *Puss in boots*, which helped me and gave me inspiration.

I would also like to thank Dr. Donald House and Dr. Timothy Davis for being my committee members. They both gave me great feedback and support.

I thank Dr. Brian Malloy for the XML parser from his Object Oriented Software Design class, which facilitated my pipeline.

I thank Kacey Coley who contributed his MeshPotato tool to the Digital Production Arts (DPA) pipeline, and who helped me to install libraries. He also introduced deep image and parallel rendering to me, which made my production easier.

I thank Samuel Casacio for his help and advice with using OpenVDB. I also thank to all the other people in the thesis meeting each week who also gave me valuable feedback and suggestions.

I thank Chen Chen for the calligraphy, which perfectly matches the theme of my animation. He and Kara Gunderson also gave me a lot reference images and videos.

I thank my parents for the financial and emotional support for doing my graduate study in the United States.

Table of Contents

Title Page	i
Abstract	ii
Acknowledgments	iii
List of Tables	v
List of Figures	vi
1 Introduction	1
2 Concepts and Design	5
3 Background	8
4 Implementation	17
5 Results	32
6 Conclusion and Discussion	38
Bibliography	40

List of Tables

4.1	Noise attributes for each layer	20
4.2	Parameters for interior and exterior lights	22
4.3	Render time comparison of single or multi-thread ray marching	25
5.1	Number of lights used for different pieces in shot01	36

List of Figures

1.1	NASA’s Sky Weather Chart[1]	1
1.2	Clouds Rendered from Commercial Packages	2
1.3	Cloud Shot in <i>A-Team</i> , Rhythm & Hues, 2010	3
1.4	Cloud Shot in <i>Puss in Boots</i> , DreamWorks Animation Studio, 2011	3
1.5	<i>Partly Cloudy</i> , A Pixar CGI animated short film, 2009	4
2.1	Storyboard	6
2.2	Screen shot of the toon shaded layout	6
2.3	Photograph of clouds from an airplane by Chen Chen	7
2.4	Cirrus example [2]	7
3.1	Data structure of VDB grids. Picture from OpenVDB website [3]	9
3.2	Levelset	10
3.3	Fractal Noise [4]	11
3.4	Nuke nodes for RGB lights	15
3.5	Cloud bunny render and compositing in Nuke Bunny grid downloaded from the OpenVDB website [5]	16
4.1	Workflow Chart	17
4.2	The process of adding displacement	19
4.3	One Cloud Sample model from vdb_view	21
4.4	Exterior and interior lights	23
4.5	Implementation of shadow gobos	24
4.6	One frame of the shadow gobo file in shot03	24
4.7	Deep images’ concept and structure	27
4.8	Deep image of cloud bunny Bunny grid downloaded from the OpenVDB website [5]	27
4.9	Example of calculating the holdout alpha channel from multiple deep images and applying it.	28
4.10	Depth calculated from deep images in shot01	29
4.11	Customized Photoshop brushes for matte painting	30
4.12	Matte painting for shot01	30
4.13	Matte painting for shot02	30
5.1	One Cloud Sample	32
5.2	One Cloud Light Pass	33
5.3	Shot01 Screenshot	34
5.4	Shot02 Screenshot	34
5.5	Shot03 Screenshot	35
5.6	All cloud clusters in shot01	35
5.7	Title Calligraphy by Chen Chen	36
5.8	Credit	37

Chapter 1

Introduction

Clouds are formed by droplets of water and ice in the sky in the low temperature environment at high altitudes. Different types of clouds are classified by their height, formation and shape characteristic (Figure 1.1). Cloud rendering is always a topic of interest for computer graphic scientists and visual effects artists because of its varied shape and whimsical movements.

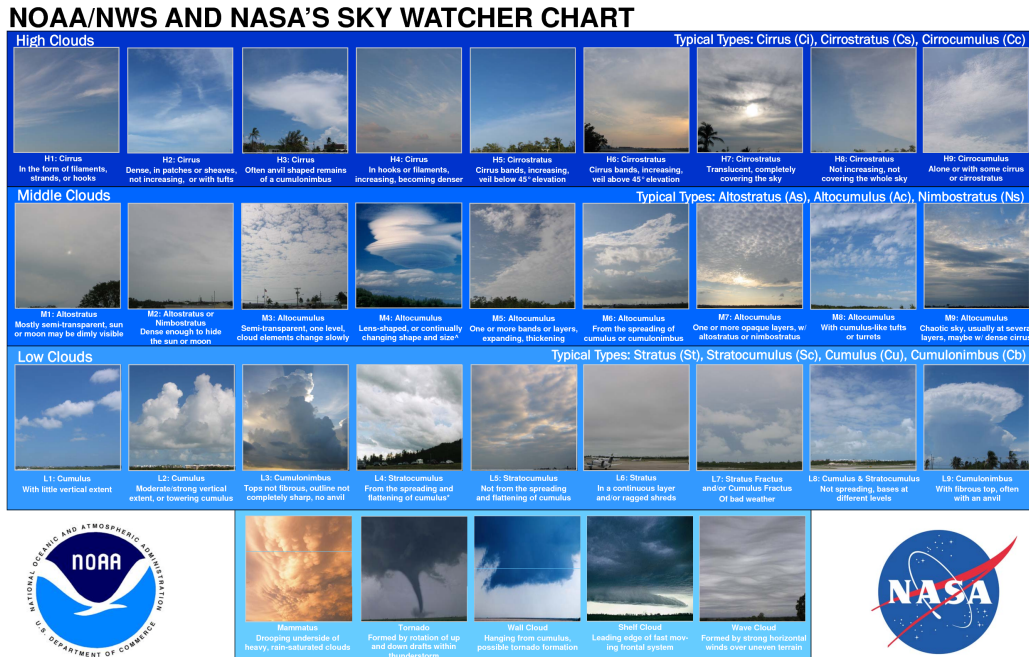
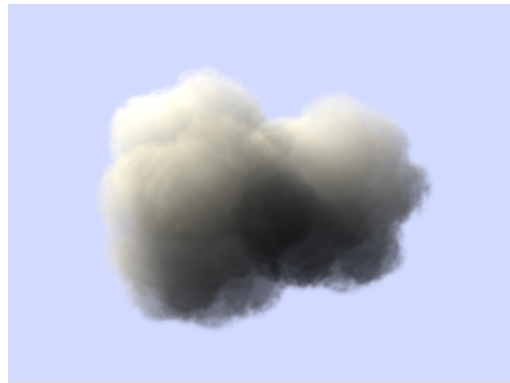


Figure 1.1: NASA's Sky Weather Chart[1]

There are several methods of generating computer-graphic clouds in featured films. For clouds that are far away from the camera, matte painting is widely used to generate backgrounds based on what the scene needs. Layered matte-painted clouds can also be animated to create an illusion of motion within the clouds. For shots that have dramatically changing camera angles and that require traveling through clouds, volumetric rendering is used for physical accuracy and visual interest. In some commercial packages such as Houdini and RenderMan, there are also methods that have been developed in order to create clouds. In a tutorial from the SideFX website [6], the Cloud Rig tool in Houdini (Figure 1.2a) is demonstrated. On the Pixar website, there is a tutorial [7] that creates clouds using a SLIM [8] shading network(Figure 1.2b).



(a) Clouds rendered using Cloud Rig in Houdini [6]



(b) Clouds rendered using RenderMan [7]

Figure 1.2: Clouds Rendered from Commercial Packages

Volumetric rendering is an accurate way to render fog, clouds, smoke, explosions and fantasy elements. Houdini has its own volumetric renderer that can handle rendering simulations of Pyro effects. Magnus Wrenninge developed the PVR System [9] for his book *Production Volume Rendering*, which is a volumetric renderer that was built in C++. Rhythm & Hues Studios developed their FELT (Field Expression Language Toolkit) [10] language for volumetric modeling and rendering which was used in feature film production like *A-Team (2010)* (Figure 1.3). DreamWorks Animation studio has developed a cloud rendering method [11] which was first applied in *Puss in Boots (2011)* (Figure 1.4), and later in *the Croods (2013)* and *How to Train Your Dragon 2 (2014)*.



Figure 1.3: Cloud Shot in *A-Team*, Rhythm & Hues, 2010



Figure 1.4: Cloud Shot in *Puss in Boots*, DreamWorks Animation Studio, 2011

Not only have clouds been used as natural elements for environments in feature films, but also they have been developed as main characters in some animations like Pixar's short animation *Partly Cloudy* (Figure 1.5). To apply this spectacular natural phenomenon as the driving element in my animation, a volumetric renderer that supports cloud modeling and rendering was designed and built.



Figure 1.5: *Partly Cloudy*, A Pixar CGI animated short film, 2009

Chapter 2

Concepts and Design

The concept of my animation came from *Inner Chapters Enjoyment in Untroubled Ease* by Zhuangzi, a Chinese Daoist [12] philosopher who lived in the 4th century BC. At the beginning of the story, it says:

“In the Northern Ocean there is a fish, the name of which is Kun, - I do not know how many li¹ in size. It changes into a bird with the name of Peng, the back of which is (also) - I do not know how many li in extent. When this bird rouses itself and flies, its wings are like clouds all round the sky. When the sea is moved (so as to bear it along), it prepares to remove to the Southern Ocean. The Southern Ocean is the Pool of Heaven [13].”

Inspired by Zhuangzi’s philosophy, I combined the concept of the ocean and the sky, which share some similarities: they are both endless, blue, connected by the horizon, and keep a large distance from each other. I illustrated Kun - the giant fish - in the sky, swimming around the sea of clouds. Because Kun is enormously large in scale and is an imaginary creature in Chinese literature, I decided to depict it by showing only its shadow on the cloud to demonstrate its size and mystery. The silhouette of a whale was used as a reference for the shape of the giant fish Kun.

Three shots were designed for my animation. After the first draft of the concept storyboard (Figure 2.1) was approved, some proxy models were placed and camera movement was designed for each shot. After the cameras were locked in Maya, a layout reel (Figure 2.2) was made in toon shading in order to display shadow movements in layout. The clouds are stationary while the shadow and cameras are animated.

¹Li is an old Chinese unit of length, 1 li equals 500 meters

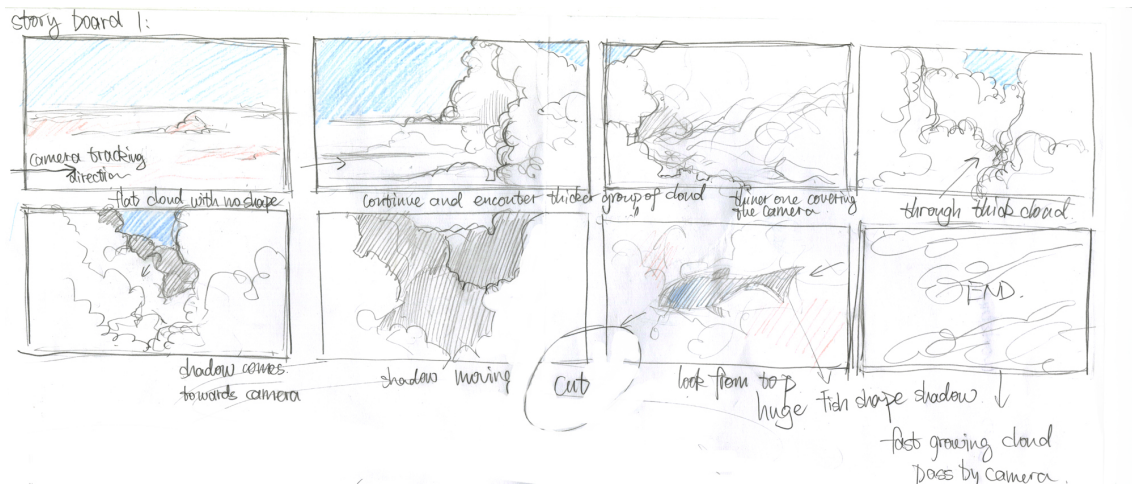


Figure 2.1: Storyboard

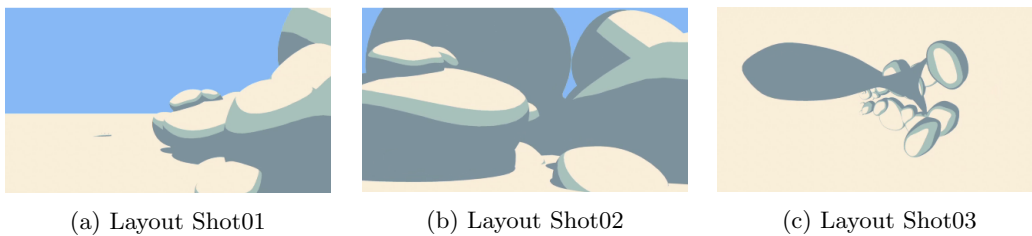


Figure 2.2: Screen shot of the toon shaded layout

There are two types of clouds in my animation - cumulus (See reference Figure 2.3) and cirrus (See reference Figure 2.4). Cumulus are low-level clouds that might precipitate. They usually have clear edges and round shapes, and play an important role in my animation. The camera is close to the cumulus and a traveling shadow of Kun is cast on them, which is a major technical challenge. Volumetric rendering is required for rendering those clouds, which provided the motivation for building the volumetric renderer. Cirrus are high level clouds that are feather-like and wispy. They serve as part of the background in the sky, and I decided to illustrate using matte paintings because they are far away and would not change its appearance as the camera moves.



Figure 2.3: Photograph of clouds from an airplane by Chen Chen



Figure 2.4: Cirrus example [2]

Chapter 3

Background

My volumetric renderer is based on a renderer I wrote in the Physically Based Effects class. The earlier renderer supported levelsets, deep shadow maps, frustum lights, Obj loading, advection and characteristic maps. However, to fulfill the need of modeling and rendering groups of clouds, the new renderer needed to be faster and more memory efficient. The desired features include:

1. OpenVDB [14] for grid storage and manipulations.
2. Faster implementation of deep shadow map (DSM) and Ray Marching.
3. Layered Fractal Sum Perlin noise implemented on OpenVDB level-set.
4. Support for rendering deep images.
5. Separate light passes for each light.
6. Python support for a user-friendly interface and flexibility.

OpenVDB is an open source C++ library that is capable of large volumetric data storage and efficient data retrieval [15]. It was developed by DreamWorks Animation and has been used for several feature films. OpenVDB is efficient because it uses a sparse tree structure (Figure 3.1). The library also has useful levelset tools that accelerate various grid manipulation, such as levelset rebuild, re-sampling, ray intersector and advection. Houdini also supports OpenVDB.

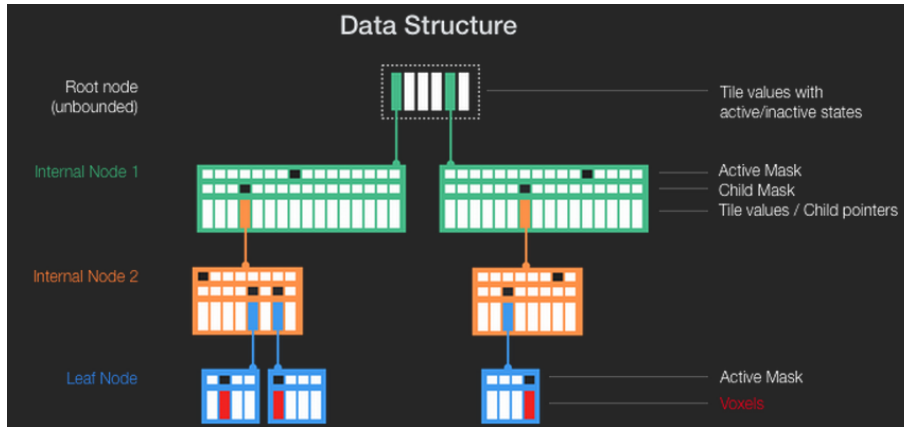


Figure 3.1: Data structure of VDB grids.
 Picture from OpenVDB website [3]

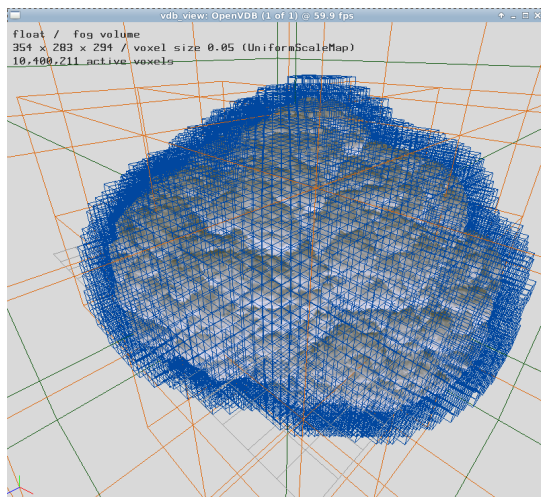
Levelsets are represented by floating point values in 3D space that indicate the distance from the point in space to the closest surface. In OpenVDB’s levelset implementation, the surface is represented by the value 0. The interior of the volume has negative values and the exterior has positive values. Dense grids are memory expensive as level sets because all grid points have values, when in practice, values are needed only near the surface. OpenVDB has a narrow band representation which saves memory and provides constant-time access to the narrow grid. Figure 3.2a showcases the structure of the grid data in `vdb_view`¹: the blue boxes are the leaf nodes, which are active voxels. The larger orange boxes are internal nodes, and the even larger green boxes are higher level internal nodes.

The narrow band is the volume where there are exact distance values that represent the levelset. An interior and exterior narrow band are defined with correspondent bandwidth. In Figure 3.2a, the blue leaf nodes outside the surface indicate the exterior narrow band. A background value is set for a levelset outside the exterior narrowband when a grid is first built. Normally, the absolute value of the background value is the `exbandwidth`² multiplied by the voxel size of the exterior narrow band. As a result, the volume is represented in a simpler and memory-efficient way. The structure of the same VDB grid is shown in Figure 3.2b. However, the precision of the grid outside the exterior and inside the interior narrow band is lost since the voxels inside the interior narrowband and outside the exterior narrowband have a constant signed value which is the background value. In the case of

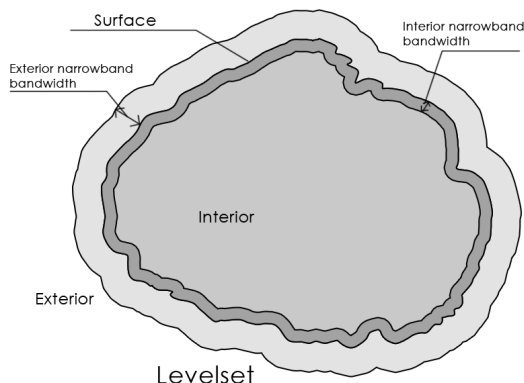
¹`vdb_view` is a commandline tool from the OpenVDB distribution that can view a VDB grid and display its internal leaf nodes.

²`Exbandwidth` is the width of the exterior narrowband

implementing noise displacement, the magnitude of which depends on the noise amplitude, a larger narrow band might be required. Details about the implementation of the layered Fractal Sum Perlin noise is discussed in Chapter 4.



(a) Levelset displayed in vdb_view, with grid nodes turned on.



(b) Structure of the same levelset

Figure 3.2: Levelset

While levelsets are efficient for storing signed distance functions and manipulating them, they do not properly represent the density of the volume. This issue occurs because the values stored inside narrowband are distances, which can be largely depending on the interior narrowband and size of the levelset, and background values inside the narrowband are signed constant. The values need to be negated and clamped between 0 to 1 to represent the density. When calculating deep shadow maps or ray marching, fog volumes were used to represent the density field, which have values between 0 and 1. In OpenVDB, the `sdfToFogVolume()` tool converts a levelset into a fog volume, in which all exterior values are set to 0, inside the interior narrowband the value is 1, and within the interior narrowband the value is clamped between 0 and 1.

OpenVDB distinguishes between a world space and an index coordinate space. An OpenVDB tree is defined in index space. A transform is associated with each grid to specify the relationship between the world space positions and the index space positions using the `worldToIndex()` and `indexToWorld()` methods. Linear and frustum transforms are currently available in OpenVDB. A frustum transform can be applied to deep shadow maps (DSM) to store lighting information, and it is also associated with a camera.

Each transform has a map that evaluates the mapping between world space and index space. Transforms can be created from maps. The implementation inside a map is a matrix that executes the transformation. For the most common situations, a linear transform for a grid is enough to represent its scale and the level of detail. However, to build a frustum grid from a camera, there is not a direct way of creating it from any transform. Instead, A frustum map must be created first from a camera, followed by a frustum transform.

Layered Fractal Sum Perlin noise was used to achieve the pyroclastic look of the clouds. Perlin noise [16] is widely used in computer graphics and the film industry because it is a procedural noise texture that varies with spatial location. Fractal Sum Perlin noise is implemented by adding several Perlin noises of different scales and parameters to achieve a more detailed and organic look [17](Figure 3.3).

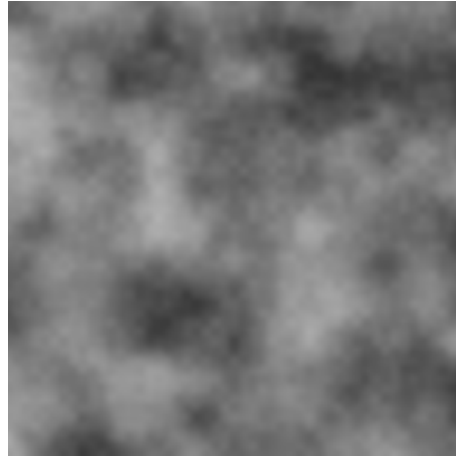


Figure 3.3: Fractal Noise [4]

In volumetric rendering, ray marching is an approximate representation of a ray penetrating the volume and returning the accumulated color back to the camera. According to Dr. Tessendorf's notes *Volumes Modeling and Rendering* [18] for the physically based effects class, the rendering equation for ray marching is:

$$L(\vec{x}, \hat{n}) = \int_{S_0}^{S_{max}} ds KC(\vec{x}(s))\rho(\vec{x}(s))e^{-\int_{S_0}^s ds' K\rho(\vec{x}(s'))} \quad (3.1)$$

$$\vec{x}(s) = \vec{x} + \hat{n}s \quad (3.2)$$

In which, the final color of the ray march is $L(\vec{x}, \hat{n})$, it is an integral from distance S_0 to S_{max} where the density $\rho(\vec{x}(s)) > 0$ along the direction \hat{n} . K is the scatter value, $C(\vec{x}(s))$ is the color at position $\vec{x}(s)$, which is a combination of emission and scatter. $e^{-\int_{S_0}^S ds' K \rho(\vec{x}(s'))}$ is a secondary ray march that accumulate the transmissivity of a light through the density from the light position towards $\vec{x}(s')$. It is time-consuming to calculate this information every step during ray marching. Therefore, deep shadow maps (DSMs) are introduced to calculate and store the light information in advance.

DSMs store the amount of light at depth in the volume. By doing a ray march from the light position towards the volume, a DSM encodes shadows that are used at render time. A DSM only needs to be calculated once for each light and volume, and can be reused if the light position and volume density do not change. DSMs reduce render time compared with calculating the secondary ray march during render time. The DSM value that is stored in each voxel is:

$$DSM(\vec{X}) = \sum \Delta s \rho(\vec{X} + \hat{N}_L s) \quad (3.3)$$

In Equation 3.3, $\rho(\vec{X} + \hat{N}_L s)$ is the density at position $\vec{X} + \hat{N}_L s$ and Δs is the length of each marching step. \hat{N}_L is the direction of the secondary ray marching. This equation is an approximation of $-\int_{S_0}^S ds' K \rho(\vec{x}(s'))$ in Equation 3.1. The exponential calculation should be executed during render time because it is not a linear function that can be stored in grid and interpolated later on.

A DSM can be stored in a rectangular grid or a frustum grid. A frustum shape can be represented as a camera frustum, which can be defined by the camera's position, view, up, right, angle of view, near and far plane.

The benefit of storing the exterior lights of a DSM in a frustum grid can be summarized as follows:

1. The closer to the light, the more detail the DSM will have.
2. When accumulating the light into the frustum grid, there is no need to do the same ray marching from every voxel in the grid. Instead, the light can be accumulated from the direction of each ray and stored into each of the frustum grid voxels.

Even though frustum grids have such benefits, they cannot be applied to interior lights. Because the frustum light shape should surround the bounding box of the object and the light position needs to be outside the bounding box. Only rectangular grids can be used for interior lights, and a lower resolution of the interior light grid would reduce memory and time.

To approximate the integral of ray marching in Equation 3.1, a step size Δs are specified.

$$\vec{x}_i = \vec{x} + \hat{n}i\Delta s \quad (3.4)$$

Along the direction of \hat{n} , the ray is marching from \vec{x} with a distance Δs for each step. For each marching step:

$$\vec{X}_{i+1} = \vec{X}_i + \hat{n}\Delta s \quad (3.5)$$

$$\Delta T = e^{(-K\rho(\vec{X}_{i+1})\Delta s)} \quad (3.6)$$

$$L = L + C(\vec{X}_{i+1})T(1 - \Delta T) \quad (3.7)$$

$$T = T\Delta T \quad (3.8)$$

ΔT represents the transmissivity in each step, T is the accumulated transmissivity, \hat{n} is the direction of the ray, $\rho(X)$ is the density at position X , $C(X)$ is the color at position X , and L is the final color. Normally, the color of the volume comes from the contribution of emission and scatter.

$$C(X) = C_{\text{emission}} + C_{\text{scatter}} \quad (3.9)$$

For clouds in which there is no emission, only scatter color is involved, so that the equation for the accumulation of the lights is:

$$C(X) = IC^S e^{-DSM(X)K} \quad (3.10)$$

In Equation 3.10, I is the intensity of the light, C^S is the material color. K is the DSM scatter coefficient value.

The OpenEXR 2.0 [19] format supports deep images. A traditional flat image usually contains red, green, blue and alpha (RGBA) channels in each pixel. A deep image has multiple color samples per pixel, each containing not only the RGBA channels but also a depth channel. The depth channel indicates the distance from that sample position towards the camera [20]. Deep.front and deep.back in each sample indicate the front and back distances of that sample. In my implementation,

only `deep.front` is stored. Deep images enable more flexibility for adjustment in post production and can calculate hold out matte in Nuke instead of during render time.

There are some changes to the ray marching process when deep images are involved. In Equation 3.5, the color and transmissivity value are accumulated along the ray. When rendering deep images, instead of accumulating, these values in each step should be stored separately in a list of deep samples along with the deep values. More details of deep image implementation and calculation will be discussed in Chapter 4.

In my cloud animation, I have several groups of clouds that are intersecting each other. Rendering all of them in one giant VDB grid is not memory efficient, and would prevent adjusting the position of each cloud individually in Nuke. Introducing deep images solves these problems.

OpenEXR uses the additive color model³ in which red, green, and blue channels are used to represent the color displayed on a monitor. In cloud rendering, several lights are used to illuminate one cloud. In the film industry, it is common practice that three lights are set to pure red, green and blue respectively to light a single volume. Lights are stored in the red, green, and blue color channels of an image. In Nuke, the color channels can be separated by shuffle node and the artist can adjust the color and intensity of each light separately. Next, the three lights can be added together to achieve the cumulative result of volume lit by three lights. Adjustment of intensity and color of different lights that is done in post production saves render time and unnecessary iterations of rendering.

³The additive color model is represented by mixing the red, green, and blue colored lights. It is used for computer screen and projectors.

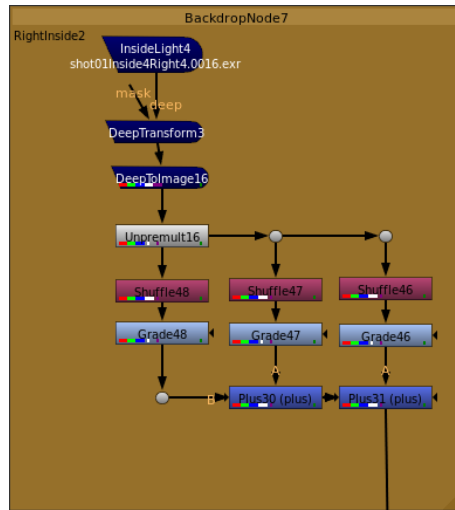
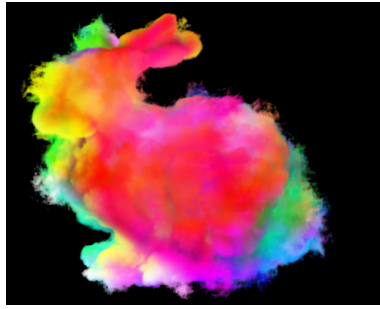
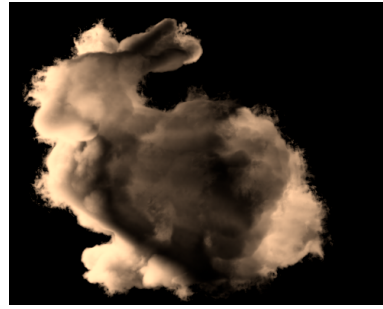


Figure 3.4: Nuke nodes for RGB lights

A simple Nuke network of color correcting a deep image rendered with RGB lights is presented in Figure 3.4. The image was converted from a deep image to a flat image. An unpremultiply node was used to get the raw color data in each channel. Three shuffle nodes, each corresponding to a color channel, were applied to separate different channels for color correction. In the example shown in Figure 3.5a, the key, fill and rim lights were stored into one image. The key light was color corrected to a warm yellow color with a higher intensity (Figure 3.5b). The fill light had a dim blue color (Figure 3.5c) and the rim light had a dim brown color (Figure 3.5d). The 3 channels were merged back into one image (Figure 3.5e). The image in Figure 3.5f resulted from applying other color corrections.



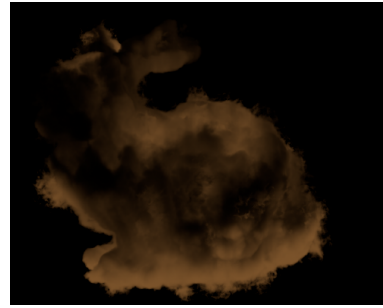
(a) Cloud bunny rendered with RGB lights



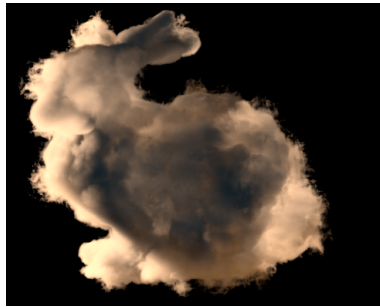
(b) Key light after color correction



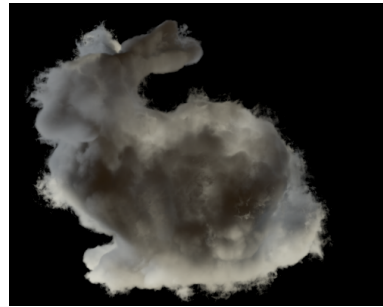
(c) Fill light after color correction



(d) Rim light after color correction



(e) Result image, added by the key, fill and rim lights



(f) Result image with another set of color corrections

Figure 3.5: Cloud bunny render and compositing in Nuke
Bunny grid downloaded from the OpenVDB website [5]

To better manage the image assets and the Nuke scripts versioning, the Digital Production Arts (DPA) pipeline was used. The DPA pipeline was developed by Dr. Tessendorf and a group of DPA students to manage assets, keep track of versions and allocate render jobs in the render farm of the DPA lab. Commercial packages like Maya, Houdini, Nuke, Mari and Blender are also integrated into the pipeline, and it is capable of handling the entire production of personal or team-based animations.

Chapter 4

Implementation

In addition to the custom renderer, Maya, Houdini and Nuke were used to previsualize the rough shape, camera movement, light distribution, and to do final compositing. The volume renderer handled three major parts: cloud modeling, lighting and rendering. The entire workflow is shown in Figure 4.1.

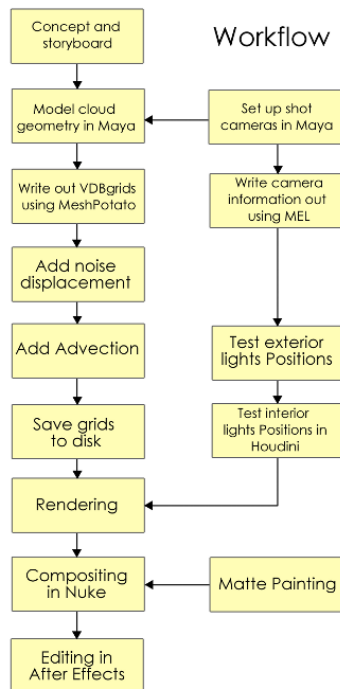


Figure 4.1: Workflow Chart

The cloud modeling process used an algorithm called “Cumulo” [21] developed by Rhythm & Hues. This algorithm has been used in multiple feature film productions. The implementation of the Cumulo algorithm on OpenVDB levelset grids is discussed in this chapter.

In order to get the pyroclastic look of cumulus clouds, 3 layers of noise displacement were applied to the base levelset of the cloud. To do that, enough exterior narrowband is needed for the displacement. `opendb::tools::levelSetRebuild()` is a function that can rebuild the levelset with a specified narrowband value. This operation can get more precise values outside the grid, or fix an existing levelset after adding the displacement to get the correct representation of the levelset. The expanded exterior narrowband from `levelSetRebuild()` was driven by the amplitude of the displaced noise and the voxel size of the grid. This relationship can be expressed by Equation 4.1:

$$\text{exteriorNarrowband} = \frac{(\text{amplitude} + \text{offset})}{\text{voxelSize}} \quad (4.1)$$

In Equation 4.1, amplitude is the amplitude parameter for the Fractal Sum Perlin noise. The offset is added to the amplitude to make sure that the displacement does not exceed the exterior narrowband. The voxelSize is `grid->voxelSize().x()`, in which, `grid` is an `opendb::FloatGrid::Ptr`, and `voxelSize()` would return a `opendb::Vec3d` which represents the voxel size in 3 axis directions.

The process of one noise displacement is as follows:

1. A levelset rebuild was implemented and the exterior narrowband was specified according to Equation 4.1. This is depicted in Figure 4.2a.

2. The active voxels of the levelset were iterated, for each voxel, `CPT_RANGE`¹ was used to find the correspondent closest surface point of the volume. The noise value on the surface point was added to the current voxel value. The surface of the volume was shifted towards the exterior area as shown in Figure 4.2b.

3. Another levelset rebuild with a small exterior narrowband was implemented to make the levelset cleaner (Figure 4.2c) and more accurate. The larger the exterior bandwidth is, the more unnecessary leaf nodes for the ray intersector² to process during ray marching. In conclusion, if any value in a grid is changed, a levelset rebuild is necessary to make the levelset more efficient and

¹`opendb::math::CPT_RANGE` in OpenVDB would find the surface point from which its position to the current voxel is the smallest.

²`opendb::tools::VolumeRayIntersector<opendb::FloatGrid>` or `(LevelSetRayIntersector)` would check if a ray hit a tile or leaf node of a volume and return the time of the start and end of the current intersection.

accurate because it is represented by the smallest possible number of nodes for the corresponding narrowband width.

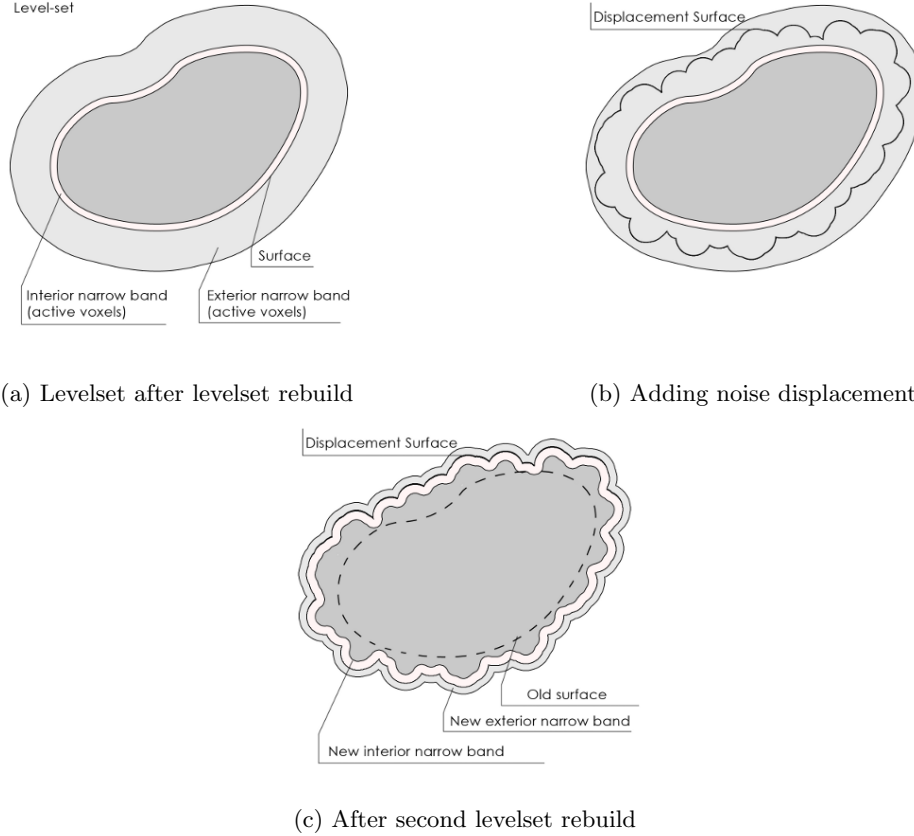


Figure 4.2: The process of adding displacement

The attributes that are used to control the Fractal Sum Perlin noise include octaves, fjump, roughness, frequency, gamma, amplitude, billow and scale. The values chosen for these parameters are shown in Table 4.1.

A second or third layer of noise can be multiplied by the first layer or second layer of cloud to achieve a billowy look in the cloud and causing smaller amplitudes in vallies of the noise, and larger amplitudes on smoother surfaces. The second layer of displacement is defined in Equation 4 [21] :

$$N_2(x) \times [clamp(|\frac{N_1(x)}{Q}|, 0, 1)]^{billow} \quad (4.2)$$

In which $N_1(x)$ is the first layer of noise and $N_2(x)$ is the second layer of noise. *billow* controls how much the amplitude of the second layer varies in the vallies or peaks of the first layer of noise. Q is a scale factor.

Layer	octaves	fjump	roughness	frequency	gamma	amplitude	billowy	scale
1	2	1	0.3	2.5	0.2	1.8	0.1	5
2	2	3.2	0.24	3.5	0.4	0.6	0.8	4
3	3.5	3.5	0.5	5	1	0.2	0.6	1

Table 4.1: Noise attributes for each layer

After 3 layers of displacement, the volume was advected to achieve the fluffy look of the cloud surface. The velocity field applied to the volume was generated by the cross product of two Fractal Sum Perlin noise with different sets of parameters, which maintain the incompressible flow of the volume and are also random enough to achieve an interesting look.

A mask can be applied to the velocity field used in the advection. In the implementation, the mask was defined as a linear interpolation between two points specified by the user, and the value was clamped between 0 and 1. With the mask applied to the velocity field, the advection on the bottom part of the cloud is more extensive than that on the top. Other kinds of mask can be applied too if the user specifies them.

A final volume for a single cloud sample viewed from `vdb_view` is shown in Figure 4.3.

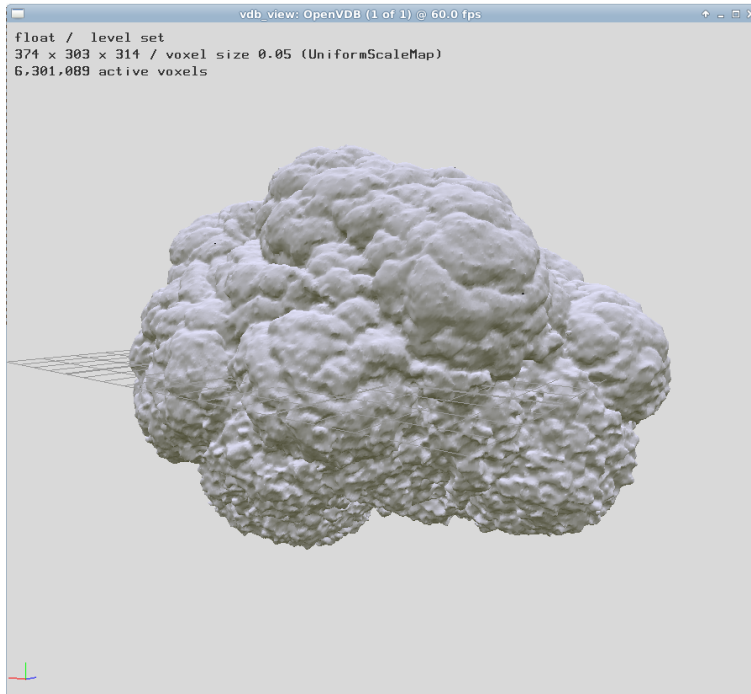


Figure 4.3: One Cloud Sample model from vdb_view

In the implementation of deep shadow maps in a rectangular grids, several issues were resolved. First, the grid for a DSM need not have the same scale as the density grid. Especially when rendering interior lights, a less detailed grid is enough to achieve a desired look. Before stamping the DSM into a grid, `openvdb::tools::resampleToMatch()` was used to re-sample the original grid into a sparser grid in which voxelsize is specified by the user.

Second, in order to get the correct density, a fog volume was used instead of a levelset grid. The reason as discussed in Chapter 3. Because of the differences in voxelsize for density and light in the grid, some voxels around the surface of the volume should have non-zero DSM, but do not because of the different interpolations of the density and DSM grid. The artifact looks like some light has leaked in the shadow area of the volume, where the interpolated values are zero by default according to the DSM grid, but the density grid has a positive value. To solve this problem, `openvdb::tools::activate()` was used first to activate all of the interior voxels of the sparser density grid, and then `openvdb::tools::dilateVoxels()` was employed to dilate the active voxels with a small width. This dilation only convert some voxels on the surface to active voxels, but does not change the actual value of those voxels. Active voxels in this slightly larger grid were used to calculated the

DSM. This method is better than an old method in order to solve the light leaking artifact. The old method would iterate every voxel and test the value of that voxel to determine if it is larger than 0 to judge if a voxel needs to calculate DSM value.

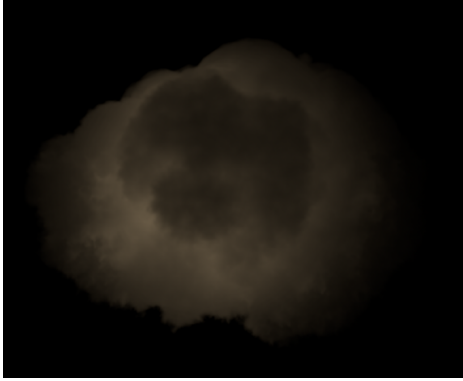
There are basically two ways to set up the light distribution to fake the multiple scattering of clouds. Both use Houdini. The interior lights should be distributed in the area where this phenomenon happens the most. The first method is to scatter some random points inside an eroded volume. The purpose of the erosion is to ensure the light position is not too close to the surface of the geometry. Lights positioned in the volume were rendered with red, green and blue color channels.

Experimentation with this method showed that the best locations for interior lights are around the areas shadowed by the exterior lights. The camera position also made some interior light positions unnecessary. In a second method, lights are placed by hand to make the process of light distribution more efficient. In Houdini spheres were created and translated into the shadow area to represent areas affected by light. These spheres were buried into the volume, and their positions were evaluated as the interior lights positions. Three lights were rendered in one image using RGB lighting, and a sequence of the interior lights were rendered according to the camera movement.

A larger scatter coefficient value (K) for DSMs would make sure that the interior lights react more sensitively towards the density of the volume than the exterior lights, which have a smaller scatter coefficient. The scatter coefficient value for ray marching should be set consistently in an XML file to ensure the same alpha channel of the image is rendered by both types of lights. Typical parameters used for different types of lights is listed in Table 4.2. Figure 4.4 displays an interior light and an exterior light with the same cloud volume. The light ray would penetrate more in the case of an exterior light than an interior light. The interior light in Figure 4.4a showcases the leak of light through the valley of the cloud geometry. Normally, multiple interior lights and three exterior lights would be rendered for each piece of cloud. In Chapter 5 there is a example of number of lights in each clouds (table 5.1).

	DSM K value	Ray march K value	Re-sampled grid voxel size (if rectangular DSM)
Exterior lights	1.0	1.0	0.1 - 0.2
Interior lights	3.0	1.0	0.3

Table 4.2: Parameters for interior and exterior lights



(a) An interior light for a sample cloud



(b) An exterior light for a sample cloud

Figure 4.4: Exterior and interior lights

In the animation story, there is a shadow silhouette of a giant fish swimming in the sky. In order to avoid modeling, rigging and animation of a fish, a gobo was used. The gobo was applied either to a frustum light or a light with rectangular DSM. When the gobo is applied to a frustum light, it projected onto the near plane of the frustum grid (Figure 4.5a). For a rectangular DSM the gobo is projected onto a user specified plane in 3D world space (Figure 4.5b). For each ray that intersects with the plane, a value on the plane texture's alpha channel will be returned for shadow information while calculating the DSMs. The shadow shape cast by the second method is more accurate and can be consistent if multiple chunks of cloud are rendered separately. The first method is faster in terms of using frustum grid to store the DSM. In addition, because different chunks of cloud might construct a different frustum grid from the same light, it is hard to keep the shadows on different chunks of cloud consistent.

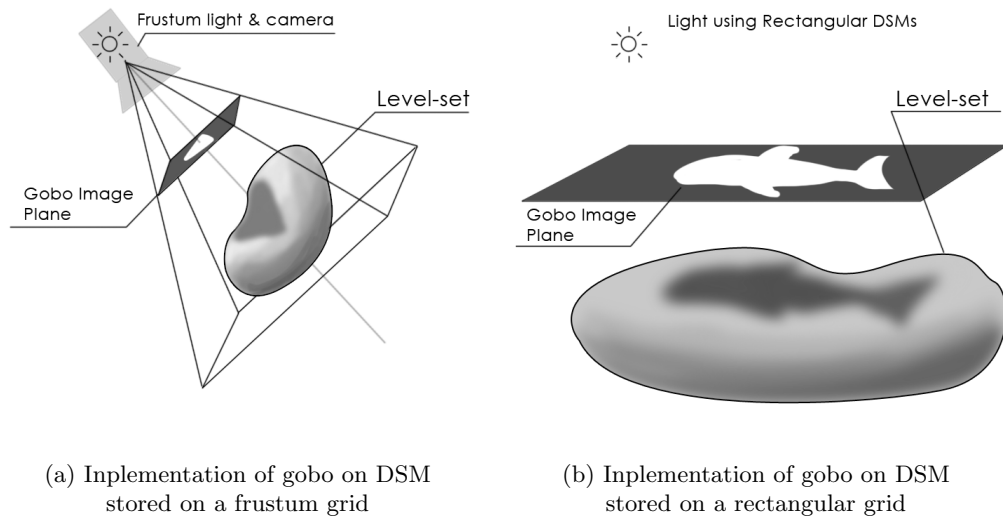


Figure 4.5: Implementation of shadow gobos

There is a close up shot in which the shadow travels on top of the clouds. A ramp that is traveling left to right was created for the shadow gobo in Nuke. The fish silhouette was hand animated in Nuke by rotopainting. A transform node and a blur node were used to make the fish shape travel slowly from right to left and blur according to the story needs. One frame of the gobo is shown in Figure 4.6.

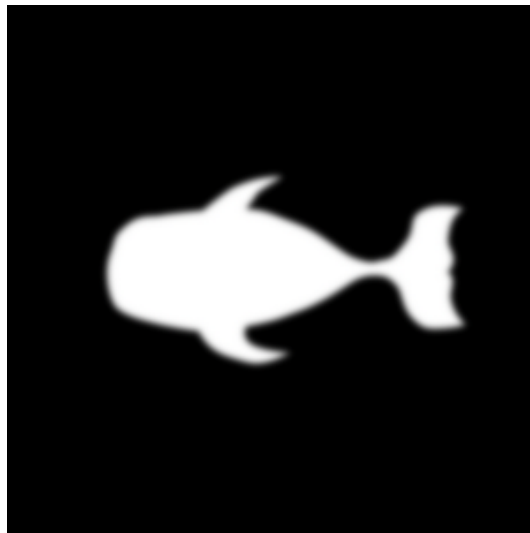


Figure 4.6: One frame of the shadow gobo file in shot03

The rendering process is time-consuming, especially for HD images. To make full use of

all the cores of the work stations in the studio, Threading Building Blocks (TBB) [22] was used to implement parallel programming in the ray marching process. Since OpenVDB uses TBB for its library, TBB was chosen for my implementation as well. A new class was defined in order to be used in the `tbb::parallel-for()` function. For each thread, one ray intersector and one grid sampler were assigned to avoid conflicts. The render time was decreased after the usage of multi-threaded rendering.

A comparison of single-threaded and multi-threaded render time is shown in Table 4.3. All images are HD (1920x1080) resolution, and they share the same render parameters, such as scatter coefficient value, ray marching step size and DSM step size. The same camera angle, light positions and cloud model were used for each image. The render time took into account all the processes required to render an image, including reading the VDB grid, building DSMs³, ray marching and exporting exr images.

No.	Hostname	Cores	Memory	Render Time	If Paralleled
1	l0003	8	16GB	24'6"	no
2	l0003	8	16GB	7'13"	yes
3	l0021	4	32GB	8'22"	yes

Table 4.3: Render time comparison of single or multi-thread ray marching

In order to have better control over the parameters and the process of rendering, Boost.Python [23] was used to expose the C++ function in Python. The whole rendering and modeling process in my renderer was controlled using different Python scripts. The library only required recompiling for major changes in the C++ source code. An XML⁴ file was used for render parameters that were not animated. For example, image resolution, ray marching scatter coefficient value and steps were stored in XML files. The parameters that are specific to each image were set in Python scripts, such as grid name, light positions, color, DSM scatter coefficient value, DSM steps and camera positions. By applying both methods, changing parameters for a render is independent from building the renderer's library.

My renderer supports deep images by using Kacey Coley's Deepimage class. The previous ray marching of a flat image stores one value for each of the RGBA channel in each pixel. The alpha

³Building DSMs is single-threaded in this case. But theoretically this process can also be multi-threaded.

⁴XML parser was from Dr. Brian Malloy's Object Oriented Software Design class.

channel is calculated by 1 minus the accumulated transmissivity along the ray. To render a deep image, the steps for ray marching were modified. For each marching step:

$$\vec{X}_{i+1} = \vec{X}_i + \hat{n}\Delta s \quad (4.3)$$

$$\Delta T = e^{(-K\rho(\vec{X}_{i+1})\Delta s)} \quad (4.4)$$

$$L = C(\vec{X}_{i+1})\Delta T(1 - \Delta T) \quad (4.5)$$

$$D = (\vec{X}_{i+1} - P_{cam}) \times V_{cam} \quad (4.6)$$

$$T = T\Delta T \quad (4.7)$$

In which, D is the value stored at the deep.front channel. P_{cam} is the camera position, V_{cam} is the normalized camera view. For each deep sample, $1 - \Delta T$ at the corresponding step defines the alpha value. Equation 4.7 is optional since the accumulated transmissivity can be calculated from the list of alpha channels in each pixel. However, a threshold of transmissivity T can still be used to stop the ray marching to decrease the image size and save memory in render time. Red, green, blue, alpha and deep.front channels construct one sample along the ray, and a list of unlimited samples would be set to represent one pixel. Figure 4.7 demonstrates the process of ray marching when rendering deep images.

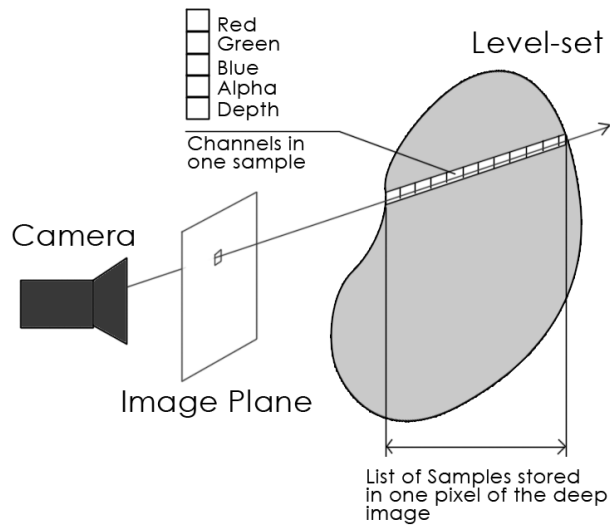
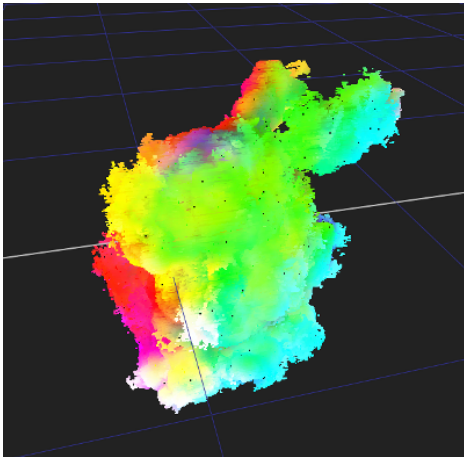
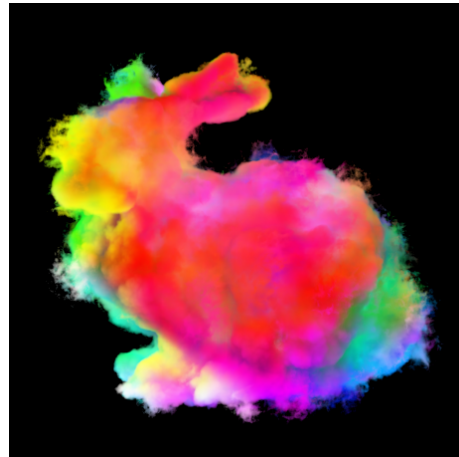


Figure 4.7: Deep images' concept and structure

OpenImageIO [24] was used to save the deep image data into an image of OpenEXR [19] format. Nuke's DeepToPoint node is a visualization tool for deep images, which converts a deep image to a 3D camera space with each sample displayed as a colored dot (See example Figure 4.8a), and the user can view the deep samples at any angle. A flat image of the same deep image is shown in Figure 4.8b.



(a) Cloud bunny, viewed in Nuke's camera space



(b) Cloud bunny image

Figure 4.8: Deep image of cloud bunny
Bunny grid downloaded from the OpenVDB website [5]

One benefit of using deep images in my show is that different pieces of clouds could be rendered separately. In Nuke, a deep merge or holdout node can calculate the holdout alpha channels based on the deep data of multiple deep images. More flexibility is also provided with the possibility of translating the volume in Nuke by applying a deep transform node. One drawback of deep image compositing is that it has long buffering time and large memory consumption. In order to save time and memory, the alpha channel that was shuffled from deep merge (hold out) nodes were cached out to image sequences and were read back in later. In this way the artist can get faster feedback from Nuke when changes are only made in color correction nodes. An example of calculating and applying the holdout alpha channel is presented in Figure 4.9.

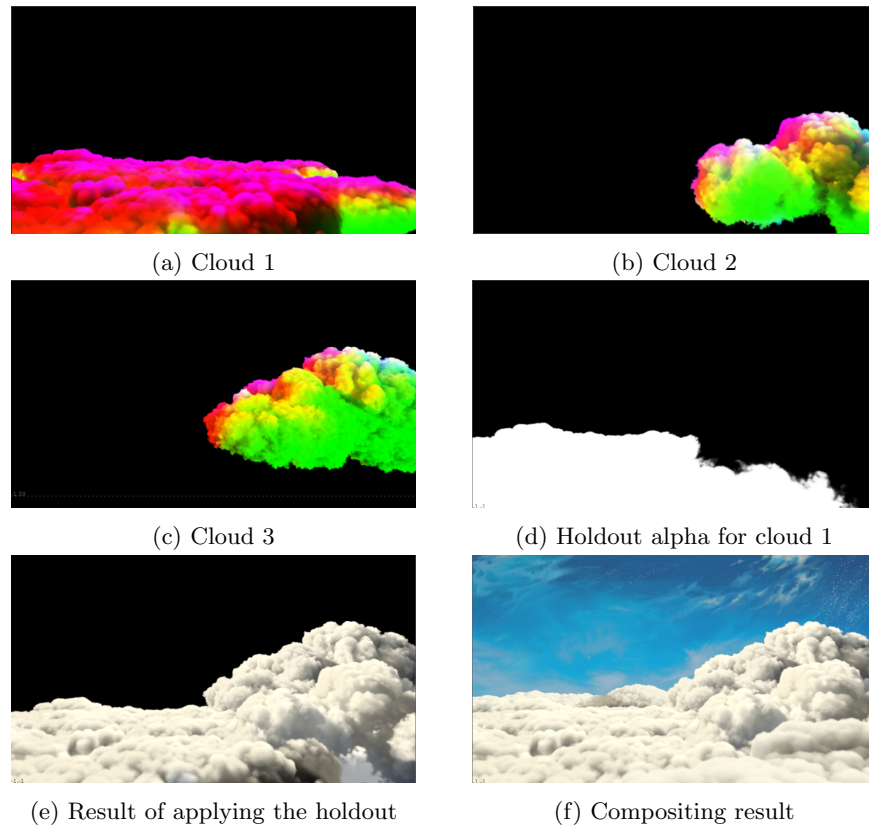


Figure 4.9: Example of calculating the holdout alpha channel from multiple deep images and applying it.

Deep data in a deep image can be used for simulating depth of field. A depth channel for a flat image can be used in a ZDefocus node in Nuke to create a controllable depth of field in post production. The depth value is between 0 and 1, which is a reverse of the deep channel⁵. A deep

⁵The deep channel stores the actual distance from camera to the point on object

image pixel contains a list of deep values calculated from the ray march process. Nuke calculates the depth channel based on the deep data when converting the deep image to a flat image.

In the process of compositing, the depth.z channel should be kept in merge nodes or premultiply nodes to keep the depth channel. Since both the ZDefocus node and depth value calculation take a long time, the depth channel was cached out in advance and read back in. In this way, the time for buffering the image was minimized and the adjustment of color made more efficient. An example of a composited depth channel is shown in Figure 4.10.

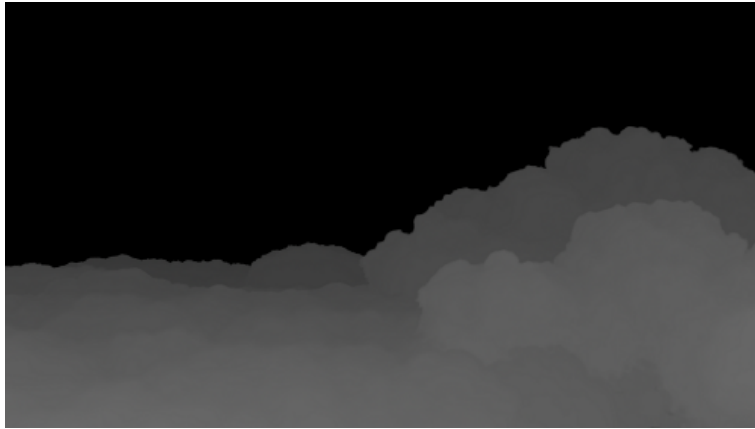
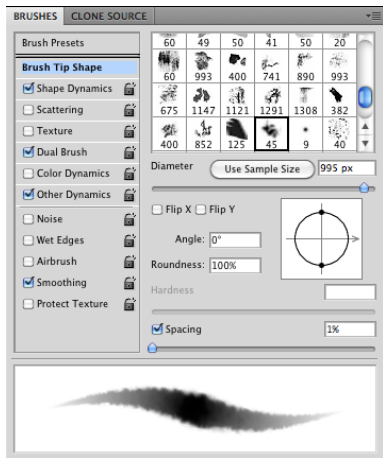


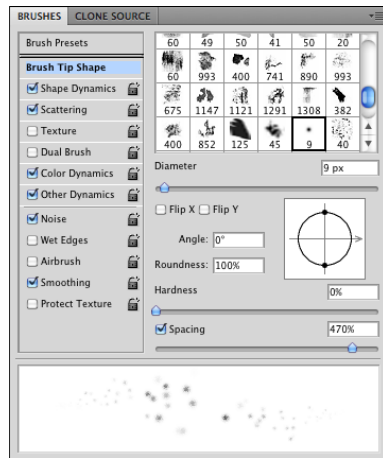
Figure 4.10: Depth calculated from deep images in shot01

The background of the sky, stars and cirrus are all matte paintings created in Photoshop from images retrieved from the Internet. See Figure 4.13 and Figure 4.12.

Some customized brushes were made to paint the fluffy feather cloud and scatter the stars from the universe. The settings for the cloud brush are shown in Figure 4.11a. A soft and organic base shape is chosen from the default Photoshop tip shape. Based on the pen pressure of the stylus for the tablet, the stroke displays different size and transparency, which provides more artistic control when painting the clouds. The settings for the star brush are shown in Figure 4.11b. The scattering feature is turned on to illustrate the star cluster. In the Color Dynamics tab, there is a slightly random variance of hue and value on each “star” it scattered.



(a) Cloud brush sample



(b) Stars Brush Sample

Figure 4.11: Customized Photoshop brushes for matte painting



Figure 4.12: Matte painting for shot01



Figure 4.13: Matte painting for shot02

The camera positions and angles were set and animated in Maya to match the layout (Figure 2.2). The first shot is a pan for the introduction of the animation from left to right. This is followed by a close up shot of cloud chunks with a shadow traveling through them towards the camera. The last shot is a birds' eye view long shot that looks down on the entire shadow shape. A MEL script was written to write out the camera position and aim for each frame and save that information out to files. In the rendering Python script, the camera information was read back in.

Chapter 5

Results

Before the actual production of my animation, I used the proposed cloud modeling and rendering pipeline and created one piece of test cloud in a turn table. The final composited image of that cloud is shown in Figure 5.1.



Figure 5.1: One Cloud Sample

The raw cloud model was built in maya and exported into a VDB grid using MPCConvert¹.

¹MPCConvert is a Maya plugin developed by Kacey Coley using MeshPotato.

Three layers of Fractal Sum Perlin noise displacement were applied to the VDB grid. Advections were also applied according to the same method discussed in Chapter 5. The final model is shown in Figure 4.3.

Figure 5.2 illustrates the steps of compositing. Three exterior lights were rendered in one image for each frame. The key light was represented by the green channel, fill light was blue, and rim light was red. The fill light was color corrected to a dim blue for the environment light from sky. The key light and rim light were set to light yellow and peachpuff accordingly for a warmer light color from the sun. Six interior lights were used to simulate the multiple scattering inside the cloud volume. Numbers 3-8 in Figure 5.2 display the effect of applying each of the interior light. The interior lights were corrected to a color that varied from light yellow to warmer yellow, in order to create contrast with the blue fill light.

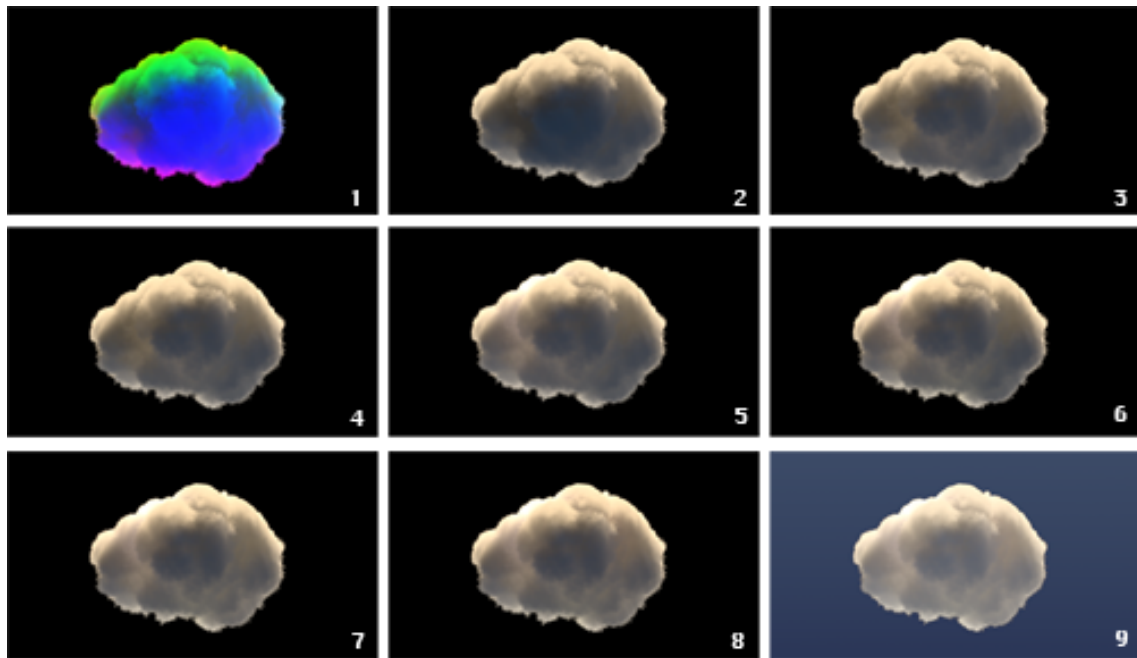


Figure 5.2: One Cloud Light Pass

After using the pipeline to create the sample cloud, larger scenes were created according to my storyboard (Figure 2.1). Several issues were encountered when applying the same method to much larger scenes. These include more run-time memory, longer render time and difficulty in compositing. Some solutions were developed accordingly, such as introducing multi-thread rendering, optimizing code and caching out intermediate images in compositing. Finished shots are listed in

Figure 5.3, Figure 5.4 and Figure 5.5.



Figure 5.3: Shot01 Screenshot



Figure 5.4: Shot02 Screenshot

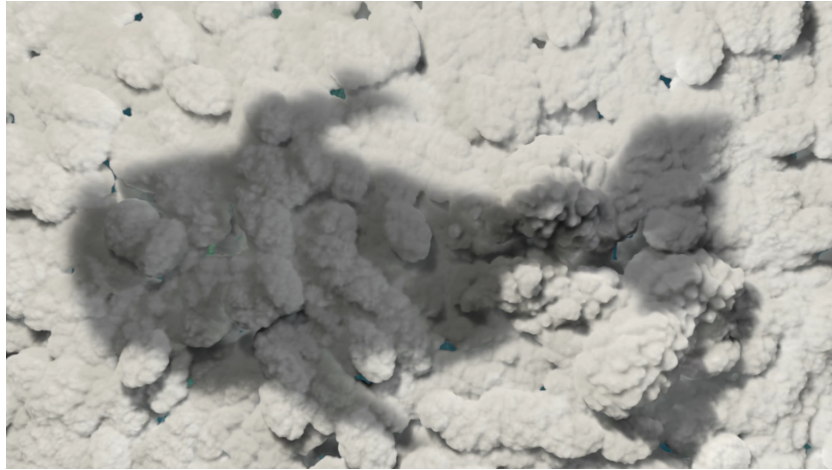


Figure 5.5: Shot03 Screenshot

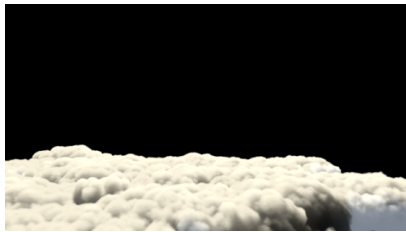
Figure 5.6 displays different clusters of clouds in shot01.



(a) Cloud 1



(b) Cloud 2



(c) Cloud 3



(d) cloud 4



(e) cloud 5

Figure 5.6: All cloud clusters in shot01

Table 5.1 shows how many exterior and interior lights were used in shot01. The cloud sea

(Figure 5.6c) has less interior lights because their thickness is not large enough to block that much sun light and create a larger shadow. Cloud 2 has less interior light because it is partially blocked by cloud 1.

	Cloud 1	Cloud 2	Cloud 3	Cloud 4	Cloud 5
Exterior lights	3	3	3	3	3
Interior lights	12	3	3	0	0

Table 5.1: Number of lights used for different pieces in shot01

The title of my animation, *Kun*, is written in the traditional Chinese calligraphy style with ink, brush and rice paper by Chen Chen (Figure 5.7). The credit of my animation (Figure 5.8) uses a similar font to match the style of the title.



Figure 5.7: Title Calligraphy by Chen Chen

CREDIT

Animation & Rendering: Zhaoxin Ye

Thesis Advisor(Committee Chair): Dr. Jerry Tessendorf

*Committee Member: Dr. Timothy Davis
Dr. Donald House*

Calligraphy: Chen Chen

Special Thanks: Kacey Coley

*Tools: OpenVDB, C++,
Maya, Houdini,
Nuke, Photoshop,
After Effects*

Figure 5.8: Credit

Chapter 6

Conclusion and Discussion

A short animation was created based on Zhuangzi's story. An OpenVDB based volumetric renderer was built to implement the displacement of Fractal Sum Perlin noise, apply advection, build deep shadow maps and manage rendering of both flat and deep images. The pipeline consisted of my own renderer implemented in C++ and Python, Maya, Houdini and Nuke, and was applied to building cloud geometries, manage lighting, rendering images and compositing the final film. The flexibility of the pipeline came from the ability to adjust camera inputs, render parameters, and batch rendering.

There are several potential improvements for the pipeline of rendering clouds as well as my volumetric render. The advection that came with OpenVDB is slow and inefficient. Characteristic maps [25], which are mapping of 3D space points from the original state of the volume to the specific moment of the advectations of several velocity grids, can largely improve the efficiency of the advection and can be reused. More complicated masks for the cloud advection could be used to generate a varied look of cloud surface. Rendering other kinds of cloud types can be explored, based on how they are formed and how dense the clouds are.

A higher resolution grid could be applied to achieve a more detailed clouds. A frustum grid, which was used for exterior lights, can also be applied to the cloud density field. DreamWorks Animation Studio has been using this method for storing the cloud geometry data in their production, which not only saves space but can also achieve a higher level of detail closer to the camera. The drawback of this method is that if the camera is changing, the frustum grid needs to be recalculated.

In terms of light distribution, the method that was used in this paper involves a lot of

manual steps. The other method investigated, using Houdini to scatter points inside the volume randomly, generated a lot unwanted points and was not efficient. If a technique could be developed to pick the scattered points based on the key light's DSM values (so that the interior light would not be generated in the light area of the key light) and the levelset value range (so that the interior lights would not be too close to the surface), that would narrow the range for selecting interior lights. The whole lighting process would be more efficient.

The cloud that I built did not involve any fluid simulation. All clouds in this project were stationary. For a real cloud, it is usually changing constantly but subtly. Simulations might be useful to achieve a flow motion for close up clouds. Clusters of clouds can also be animated to mimic formation and splitting to achieve a more dynamic look.

Bibliography

- [1] K. Laman, L. Hilbert, J. Lee, and NASA. http://science-edu.larc.nasa.gov/cloud_chart/.
- [2] Cirrus, http://commons.wikimedia.org/wiki/file:cirrus_over_warsaw,_june_26,_2005.jpg.
- [3] OpenVDB About, <http://www.openvdb.org/about/>.
- [4] Fractal noise, http://en.wikipedia.org/wiki/perlin_noise.
- [5] Cloud model download, <http://www.openvdb.org/download/>.
- [6] Cloud FX in Houdini, http://www.sidefx.com/index.php?option=com_content&task=view&id=2498&itemid=261.
- [7] Cloud rendering in renderman, <http://renderman.pixar.com/view/volume-clouds>.
- [8] SLIM, Pixar's RenderMan Studio Shader Tool, <http://renderman.pixar.com/view/slim-shader-tool>.
- [9] M. Wrenninge. *Production Volume Rendering Design and Implementation*. 2013.
- [10] M. Wrenninge, N. Zafar, J. Clifford, G. Graham, D. Penney, J. Kontkanen, J. Tessendorf, and A. Clinton. Volumetric methods in visual effects. *SIGGRAPH 2010 Course Notes*, pages 287–296, 2010.
- [11] B. Miller, K. Museth, D. Penney, V. Kulathumani, and N. Zafar. Cloud modeling and rendering for puss in boots. 2012.
- [12] Daoist, <http://en.wikipedia.org/wiki/taoism>.
- [13] Z. Zhuang. Enjoyment in untroubled ease, <http://ctext.org/zhuangzi/enjoyment-in-untroubled-ease>.
- [14] K. Museth. OpenVDB, <http://www.openvdb.org/>.
- [15] K. Museth. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics*, 32:23, 2013.
- [16] K. Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.
- [17] N. Blevins. Fractal noise http://www.neilblevins.com/cg_education/fractal_noise/fractal_noise.html.
- [18] J. Tessendorf. Volume modeling and rendering. 2014.
- [19] OpenEXR, <http://www.openexr.com/>.
- [20] F. Kainz. Interpreting OpenEXR Deep Pixels. 2013.

- [21] M. Wrenninge, N. Zafar, J. Clifford, G. Graham, D. Penney, J. Kontkanen, J. Tessendorf, and A. Clinton. Volumetric methods in visual effects. *SIGGRAPH 2010 Course Notes*, pages 95–104, 2010.
- [22] Threading Building Blocks, <https://www.threadingbuildingblocks.org/>.
- [23] Boost.Python, http://www.boost.org/doc/libs/1_55_0/libs/python/doc/.
- [24] OpenImageIO, <https://sites.google.com/site/openimageio/home>.
- [25] J. Tessendorf and B. Pelfrey. The characteristic map for fast and efficient vfx fluid simulations.