8-2012

THE EXPLOSION EFFECT: A Custom Volume FX Production Pipeline

Hongyuan Jia
Clemson University, hongyuj@g.clemson.edu

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

Part of the Fine Arts Commons

Recommended Citation
https://tigerprints.clemson.edu/all_theses/1435

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.
THE EXPLOSION EFFECT: A CUSTOM VOLUME
FX PRODUCTION PIPELINE

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
Hongyuan Jia
August 2012

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Donald House
Dr. Brian Malloy
Abstract

Explosions are very important elements in film FX production. This paper introduces a custom volume FX production pipeline and how to use the pipeline to produce explosion FX. A detailed explanation is given of the pipeline artistic and technical aspects.
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
</tr>
<tr>
<td>Abstract</td>
</tr>
<tr>
<td>List of Figures</td>
</tr>
<tr>
<td>1 Introduction</td>
</tr>
<tr>
<td>1.1 Explosion FX production history</td>
</tr>
<tr>
<td>1.2 Artistic goals of the explosion FX</td>
</tr>
<tr>
<td>1.3 Production solution</td>
</tr>
<tr>
<td>2 Volume FX pipeline</td>
</tr>
<tr>
<td>2.1 Volume Rendering Program(VRP)</td>
</tr>
<tr>
<td>2.2 Fluid Simulation Based on SELMA</td>
</tr>
<tr>
<td>2.3 Side Effects Houdini</td>
</tr>
<tr>
<td>2.4 Pipeline Strategy</td>
</tr>
<tr>
<td>3 Background</td>
</tr>
<tr>
<td>3.1 Ray Marching Algorithm</td>
</tr>
<tr>
<td>3.2 VEX/VOP SOP</td>
</tr>
<tr>
<td>3.3 SWIG</td>
</tr>
<tr>
<td>4 Implementation</td>
</tr>
<tr>
<td>4.1 Grid data structure</td>
</tr>
<tr>
<td>4.2 SELMA applied to fluid simulation</td>
</tr>
<tr>
<td>4.3 Visualize grids data in Houdini</td>
</tr>
<tr>
<td>4.4 Use SOPs to process grids data in Houdini</td>
</tr>
<tr>
<td>4.5 Particle animation</td>
</tr>
<tr>
<td>4.6 Particle based pyroclastic volume</td>
</tr>
<tr>
<td>4.7 Match VRP camera with Houdini’s</td>
</tr>
<tr>
<td>4.8 Post-production</td>
</tr>
<tr>
<td>5 Results</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Volume FX in Battle Los Angeles</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Fireball pattern of explosion [1]</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>”Mushroom” shape of explosion [2]</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>The pipeline strategy</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Ray Marching</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Explosion FX Production Pipeline</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Sparse grid data structure</td>
<td>16</td>
</tr>
<tr>
<td>4.3</td>
<td>Grids data visualization</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Display options menu about point attribute</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>The pipeline of processing grid data</td>
<td>21</td>
</tr>
<tr>
<td>4.6</td>
<td>Source vector grids for merging</td>
<td>22</td>
</tr>
<tr>
<td>4.7</td>
<td>Merging grid data tests based on 3 arithmetics</td>
<td>22</td>
</tr>
<tr>
<td>4.8</td>
<td>Merging tests for fluids simulation velocity fields</td>
<td>23</td>
</tr>
<tr>
<td>4.9</td>
<td>VOPs netwrok of MergeGrids operator</td>
<td>24</td>
</tr>
<tr>
<td>4.10</td>
<td>Particle animation advected by velocity field</td>
<td>25</td>
</tr>
<tr>
<td>4.11</td>
<td>Modify particles using Lattice SOP</td>
<td>25</td>
</tr>
<tr>
<td>4.12</td>
<td>Pyroclastic volume definition</td>
<td>27</td>
</tr>
<tr>
<td>4.13</td>
<td>Pyroclastic sphere with different parameters</td>
<td>27</td>
</tr>
<tr>
<td>4.14</td>
<td>Particle based pyroclastic volume</td>
<td>28</td>
</tr>
<tr>
<td>4.15</td>
<td>Houdini camera FOV</td>
<td>31</td>
</tr>
<tr>
<td>4.16</td>
<td>Camera definition in volume rendering program</td>
<td>31</td>
</tr>
<tr>
<td>4.17</td>
<td>The comparison between the source image and the compositing image</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>The explosion FX</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

An explosion is a rapid increase in volume and release of energy in an extreme manner, usually with the generation of high temperatures and the release of gases [9]. Explosions are an important FX element that greatly influences the story and atmosphere of movies. Historically filmmakers have invented many ways to produce explosions for films. Aside from computer graphics, explosions are usually generated by pyrotechnic machines. However, more and more explosion elements are produced by computers as the quality of the computer graphics techniques improves [3]. The techniques to produce FX based on computers are usually referred to as computer generated imagery, or CGI.

1.1 Explosion FX production history

Before CGI was widely used in the film industry, explosion FX was mainly generated by the physical methods, like powder or pyrotechnic machines. Traditional explosion techniques have many limitations and shortcomings. First, traditional methods are dangerous. Some explosion FX were created by chemical powders,
Figure 1.1: Volume FX in Battle Los Angeles
Copyright 2011 Sony Pictures Digital Inc. All Rights Reserved

Figure 1.2: Fireball pattern of explosion [1]
which actually resulted in many accidents. Second, physical explosions are hard to control, so some specific explosion effects are very difficult or impossible to generate. The explosion usually comes out with random shapes and motions. It is very difficult to create explosions with a specific shape or motion. Figure 1.1 shows an explosion form that is impossible to reproduce with physical FX methods. In addition, some large-scale explosions, such as nuclear bomb, are impossible to do by physical methods without violating international treaties. Third, generating explosion FX by physical methods is prohibitively expensive. Finally, physical explosions need to get governmental approvals and file a lot of paperworks, which make it hard to work on schedule. In contrast, the explosion FX based on CGI solves all of these problems and addresses limitations mentioned above. It is not dangerous any more, because everything is done virtually on computers. Besides, the new technologies make the
FX production cheaper to a certain extent. The CGI explosion does not need to really destroy anything physically, and some FX shots can be easily duplicated digitally. Due to these two advantages, CGI explosions have improved film visual effects and made the budgets of FX production relatively lower. The artists can design and create the explosion FX with any forms and shapes without worrying if the effects obey the physical laws or limitations. The volume FX pipeline discussed in this paper is based on the current CGI technology.

1.2 Artistic goals of the explosion FX

From visual standpoint, explosion FX is a mixture of fireballs and smoke with turbulently buoyant movements. In production, explosion FX have to meet many artistic requirements. The essential requirements are controllable color and shape, specific movement and image compositing. Controllable color means that the volume can be set with variable colors in specific areas frame by frame. In this project, bright fireballs need to gradually become dark smoke while losing energy and temperature. The smoke volume transforms into a ”mushroom” shape. Figure 1.2 shows a fireball pattern and Figure 1.3 shows a ”mushroom” volume shape in a real explosion. These are the references for this project. The specific movement means that the volume motion in space is not totally random changing or dependent upon the physical simulation. Direct manipulation must provide specific patterns. The explosion needs to emphasize curling motion as a display of explosive power. All of these requirements are designed for enhancing the artistic impact for the audience. In summary, the explosion FX should look naturally real, but still be controllable in order to follow the artistic designs.
1.3 Production solution

A flexible and interactive FX pipeline is built for satisfying the artistic requirements of this explosion FX. The pipeline is a combination of a custom-written volume rendering program and Houdini, which provides the creative methods to set the volume color, shape and movement. The volume color and shape are mainly controlled by Houdini’s particle system. The volume movement is driven by fine-tuned velocity fields, which mix the procedural simulation and manual modifications in Houdini. In addition, the pipeline has functions to interactively set FX parameters and quickly preview the FX animation. The pipeline is an interactive and controllable FX production platform, which provides effective ways to fulfill the FX design.
Chapter 2

Volume FX pipeline

The volume FX production pipeline in this project is split into two major parts: C++ code and custom Houdini nodes. The C++ program is a volume renderer based on the Ray-marching approach. The program includes volume modeling, rendering and fluid simulation. Volume modeling includes several types, such as implicit volumes, Wisps, pyroclast, Stamped Noise and particles. The fluid simulation module is based on an incompressible Navier-Stokes fluid solver to simulate the smoke effects. The C++ code is compiled into a library and a Python module through SWIG. Rendering configurations are set in Python scripts. This working mode is very flexible and effective and makes it possible to integrate with Houdini. The Houdini nodes were usually created with Python SOP and VEX/VOP types. The SOPs are responsible for exchanging data between Houdini and the rendering program; the VEX/VOP nodes are used to process simulation grid data, such as velocity, density and color fields. This pipeline lets the user produce volume FX in an interactive working environment. For example, the user can set camera animation, visualize or re-process simulation data frame by frame in Houdini. In addition to being controllable and flexible, this pipeline can exploit Houdini’s comprehensive
particle system. The following sections will introduce the essential contents of this pipeline.

### 2.1 Volume Rendering Program (VRP)

Volume rendering usually is designed for producing some kind of translucent materials, such as fog, smoke, fire and clouds. Due to the translucent features of volume materials, the rendering algorithm differs from traditional surface material rendering. Translucence means that entire materials inside of geometry influence the color of each pixel during rendering. Ray Marching is used for volume rendering. The rendering algorithm computes the lighting accumulated over a ray vector that traverses a volume space. Volume geometries includes Pyroclast, Stamped Noise, Wisps, Levelsets and volumes based on particles. A grid data structure is used in this rendering program, because the Ray Marching process is time-consuming, especially in rendering Deep Shadow Maps (DSM). In addition to improving rendering performance, the grid data structure is also essential for some type of volume geometry. For instance, Levelsets are totally dependent on the grids to define the density fields for final rendering. The chapter 4 will discuss the detail of this program.

### 2.2 Fluid Simulation Based on SELMA

SELMA [6] (SEmi-LAgrangian MApping) is an effective method to store and retrieve the simulation data based on a gridded mapping technique. It combines the advantages of gridless and gridded volume simulation approaches. In this project, the simulation module with SELMA created the ”mushroom” shape of explosion and buoyant effect.
2.3 Side Effects Houdini

Houdini is a high-end 3D animation package developed by Side Effects Software headquartered in Toronto, Canada. It is the successor to the PRISMS ecosystem of tools. Its chief distinction from other packages is that it has been designed as a purely procedural environment[10]. This commercial software is widely used in the FX industries. Its outstanding features includes its clear node based environment, high customizability and a powerful programming interface. For instance, Houdini provides several ways to define new operations. Houdini is equipped with several programming interfaces, like Hscript, VEX, C++ API and Python API. In this project, the Python API and VEX operators have been extensively used. In addition, Houdini provides a convenient interface to access low-level data. These characteristics make Houdini an excellent tool for this project.

2.4 Pipeline Strategy

The pipeline is split into two parts: the C++ code for VRP and the custom Houdini operators. The C++ code handles volume rendering and fluid simulation; the Houdini operators handle processing the simulation data, generating particles and exchanging data between Houdini and VRP (only Python SOPs). Figure 2.1 shows the pipeline strategy. The programming procedures includes:

- Compile the VRP Python module using SWIG
- Write the interface Python code (glue layer) to repackage the Python module in order to access it via Houdini operators
- Define the Python SOPs and VEX SOPs
• Write Python scripts to configure the rendering parameters and environments

This strategy brings two benefits to the volume pipeline: custom volume rendering and simulation algorithms and an interactive working environment: the volume appearance is rendered by the customized algorithms; meanwhile, the rendering camera, simulation data and volume parameters are interactively set in Houdini using expressions and key-framing.

Figure 2.1: The pipeline strategy
Chapter 3

Background

3.1 Ray Marching Algorithm

Ray Marching computes images from 3D volumetric data sets [11]. Figure 3.1 shows the Ray marching algorithm, which computes final rendering color based on 3D volumes instead of surfaces on which traditional shaders compute a final rendering color. The algorithm for ray marching in volume rendering is a numerical approximation of light $L(xc, np)$ that is received by a camera located at position $xc$, at the pixel that is on the sampling ray vector. The rendering equation accumulates light emitted by the volume along the line between $np$ and $fp$. The rendering equation in this context is a simplified single-scatter approximation of the fuller theory of radiative transfer:

$$L(xc, np) = \int_0^\infty ds \; C^T(x(s)) \; \rho(x(s)) \; e^{-\int_0^s ds' \kappa \rho(x(s'))}$$

$xc$ is camera location; $ds$ is the size of each Ray Marching step; $np$ is the start point of Ray Marching and $fp$ is the end point; $s$ is the whole distance of numerical integration for each pixel; the parameter $x(s)$ is a straight line path originating at the camera and
moving outward along the sampling ray vector to points in space a distance $s$ from
the camera; $\rho$ is a material property of volume in space; $\kappa$ is a constant controlling
the volume opacity overall during rendering process; $C^T$ is the color of the volume at
each sampling point. [7].

![Figure 3.1: Ray Marching](image)

### 3.2 VEX/VOP SOP

VEX is the abbreviation of "vector expression language" and is an embedded
scripting language of Houdini. VEX’s syntax is similar to the C programming lan-
guage. It also absorbed many concepts from Pixar’s RenderMan Shading Language.
VEX scripts are automatically compiled by Houdini before it is executed, so it is
efficient in processing the huge number of data points. VOP is the GUI version of
VEX. Each VOP node includes VEX code.
In Houdini, every operator belongs to a specific data type. SOP is the abbreviation of Surface Operator, a geometry data type. SOPs describe and process surface geometry data, such as point, primitive, edge and so forth. Custom SOP nodes have two types: Python based and VEX/VOP based. A VEX/VOP SOP is designed to process the surface geometry data by VEX codes or VOPs.

### 3.3 SWIG

SWIG (Simplified Wrapper and Interface Generator) is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code [4]. SWIG automatically generates Python wrapper code according to SWIG interface files, which are generated from C++ header files. This means the programmer does not need to change their C/C++ code for compiling into a Python module. The only thing needed is SWIG interface files for C/C++ classes and functions. In this project, the C++ code is compiled into a Python module by this software.
Chapter 4

Implementation

The final explosion effect consists of multiple layers of volume rendering, which were created by several volume modeling techniques. Figure 4.1 shows the basic working procedures of this production pipeline:

1. Use the fluid simulation module to simulate the explosion effect and write out the simulation results as grid data, which include the velocity and density fields.

2. Import the simulation results into Houdini to visualize the density and velocity fields in order to analyse the motion.

3. Employ custom SOP and VEX nodes to modify the velocity data for fine-tuning the motion.

4. Use the custom Python SOP WriteGrid to export the velocity fields data from Houdini to sparse grids files.

5. Import the final velocity fields into Houdini and use them to advect Houdini particles.
6. Set points attributes, such as pscale, opacity, color, Perlin Noise parameters etc, for particles and then write the Houdini particles into a sequence of GEO files.

7. Import the particles GEO sequence into the custom volume rendering program to generate the particle based pyroclastic effect and render a sequence of EXR images.

8. Set compositing, glow effect and color correction in NUKE.


The following sub-sections discuss the essential implementations and concepts in detail.

4.1 Grid data structure

A sparse grid has been extensively used in this project. The grid data structure is a very important implementation. It is related to volume modeling, rendering and data exchange. First, the application of grids improves the program performance. Second, some volume geometries are completely defined by gridded data. In addition, data exchange between VRP and Houdini depends upon the grid data files. So the grid data structure is a core of pipeline.

4.1.1 Grid data structure and volume modeling

The volume modeling method has two types: implicit function based and gridded fields based. The former is defined by implicit functions. For example, a sphere can be defined by an implicit function \( f(x) = |x - \text{center}| - \text{radius} \), which consist of a radius and a center vector. The sphere volume is defined in the function
Figure 4.1: Explosion FX Production Pipeline
range, where $f \geq 0$. The implicit function based volume is resolution independent, but it is inconvenient to build natural and free form shapes and consumes a huge rendering time when the scene consists of many implicit volume geometries. To solve this problem, the density and color information of volumes can be sampled onto 3D grids, which accelerate rendering. The density and color data on grids are called density field and color field. The grid data can be of several types, including scalar, vector and matrix.

### 4.1.2 Grid data structure and program performance

The grid data structure boosts the program performance. Volume rendering and modeling of implicit functions involves heavy numerical integration in 3D space. So to reduce the sampling number and computation complexity, it is valuable to use grids. Sampling the volume data onto grids is a critical method of applying grids. DSM (Deep Shadow Map) rendering is a typical example. Grids make volume rendering with DSMs possible with acceptable time consumption.
4.1.3 Improvement of grid data structure

Due to the importance of the grid data structure, the sparse grids with binary I/O was applied to this project for improving the data access and storage. Figure 4.2 shows the customized sparse grid data structure, which includes two layers of grid data. The first layer is defined by a STL vector container, which stores participation block (PB) addresses. The second layer stores the actual grid data, including float, Vector (a custom 3D vector type) and Color types. Each PB has a marker to indicate if this block stores valid data. Therefore, only blocks with a valid marker reside into the memory and are exported to data files. And the program will also automatically ignore the invalid blocks during Ray Marching process. In addition, binary I/O of grids improves the data I/O speed and saves memory. A file format was designed to store grid data with a variety of types. The grid files include two parts: file head and data block. The file head stores the sparse grid structure information, like dimension, span between grid points, data type and the markers of each participation block. The data blocks store the actual memory data in continuous memory. When the program reads data from a file, it first rebuilds the grid data structure according to the file head, and then reads the valid data blocks based on the marker list and puts each data block into the corresponding block memory address. As a result of the sparse grid design, the program is greatly improved in performance, I/O speed and rendering.

4.2 SELMA applied to fluid simulation

The "mushroom" shape of the explosion and buoyant effect in this project are created by the fluid simulation module, which is based on Navier-Stokes equation. The following is the basic simulation procedures for each time step:
1. Apply the buoyant force: velocity += -gravity * density
2. Advect velocity by itself: advect( velocity, velocity, dt)
3. Use the FFTW library to process the velocity grids in order to get a divergence-free velocity field.
4. Write out velocity field data.
5. Compute SELMA displacement data from velocity field.
6. Use the SELMA displacement to remap the density fields for next simulation step.

The fluid simulation of the density can be based on gridless or gridded data. The gridless simulation is resolution independent. However, it costs more rendering time, because the advection iteration linearly increases in executing time. The gridded simulation has lower cost, but the final structure depends upon the grid resolution. In this project, a compromise, called SEmi-LAgrangian MApping (SELMA), was applied for the smoke effect [6]. Rendering with SELMA generally outputs relatively higher detail than rendering with gridded simulation data. The basic idea is that the simulation program will generate the displacement vector fields for density and color fields at each simulation step. Then the displacement data will be used as the volume advection fields to remap volume density and color during rendering process. Since SELMA stores the movement information of each step instead of the final density data, the simulation avoids losing volume density from numerical dissipation at each simulation step.

In addition, the velocity fields were written out as files during simulation. Then they were used in Houdini to drive the particles to move. The particles are the source of pyroclastic volume, which generated the final smoke image.
4.3 Visualize grids data in Houdini

Gridded data can be converted into geometry data using Python SOPs. Then the converted data can be displayed through Houdini native point visualization functions. Figure [5] is the visualization example for scalar and vector grids data. Figure 4.4 shows the setting menu of display options. This visualization technique gives artists a real-time feedback of the simulation result without time-consuming rendering. The fine-tuning process of smoke motion benefits from this technique.
4.4 Use SOPs to process grids data in Houdini

Houdini SOPs were used in this pipeline to process gridded data. This process refined the explosion in various aspects. For example, the velocity field of explosion has been modified in order to emphasize the curling motion. The key step of processing grid data is to covert the grid data into Houdini native point data. After the conversion is done, the grid data will be valid for any SOP. Figure 4.5 shows the basic processing pipeline, which includes the following procedures:

1. Use a Python SOP node to load grid data and convert them into point data.
2. Modify the point data by SOP nodes.
3. Reset the grid value with the modified point data.
4. Write the grid data back into grid files.

The following subsections will explain the grid data conversion and two approaches of modifying grids data: merging grids and modifying grids by Houdini native SOPs.

4.4.1 Convert sparse grid to Houdini SOP data type

A custom Python SOP was designed for converting sparse grid data into Houdini point data. This operator creates Houdini geometry points according to imported grid dimension information and stores the grid data onto geometry points. Houdini point attributes can be set as float, array or other data types. The float attribute usually represents float grid data, like the density field. In a similar way, array attribute stores vector or matrix grid data, such as velocity field or SELMA displacement. The Python code for the conversion process is attached in the appendix A.
Figure 4.5: The pipeline of processing grid data
Figure 4.6: Source vector grids for merging

Figure 4.7: Merging grid data tests based on 3 arithmetics
4.4.2 Merging grids

Merging grids is a typical operation that is done through SOPs. The working mechanism is similar to the mixing operation between two image layers in Adobe Photoshop. Photoshop computes the final image color pixel by pixel, while MergeGrid operator computes the final grid data point by point. In the merging implementation, a VEX SOP node is responsible for computing the point data, because VEX is highly efficient in processing huge number of points. The merging has several steps: first, a Python SOP node reads the grid data from binary or ASCII grid files. second, create the grid points based on the imported grid information. third, transfer each grid point’s data into point attribute. For example, density data will be stored into a float point attribute; the vector field or color field will be stored in an array point attribute. The Houdini Object Model (HOM) API provides the functions to do
these tasks. After the grid geometry (which only includes a bunch of points without primitive or polygonal mesh) has been built, a VEX node does the computation for the two sources grids. For example, an additional calculation between a vector grid with constant up vector and another vector grid with turbulence vector will generate a new turbulent vector grid with a up direction. Figure 4.6 and Figure 4.7 show the basic merging effects. Finally, another Python SOP exports the final grid into a binary file, which is later used in rendering or simulation procedures (Figure 4.8 is a merging test of velocity field). This merging technique between two grids can be any arithmetic calculation, such as addition, subtraction, multiplication, division or other conditional calculation. The processing is designed to specific art requirements.

4.4.3 Modifying grids using native Houdini SOPs

After grid data are converted into native Houdini points, they are ready for any SOP, such as: Point, Color, Painting and so on. There are many handy tools
to modify grid data for reaching artistic goals. In this project, the Point and Color SOPs were used to visualize the fluid simulation result and set the turbulence fields.

4.5 Particle animation

![Figure 4.10: Particle animation advected by velocity field](image)

![Figure 4.11: Modify particles using Lattice SOP](image)

Particle animation is an essential component of this project, since the animated
particles are the source of the pyroclastic volumes. Several custom SOPs and POPs have been set in this pipeline for fine-tuning the particle animation in order to enhance the visual effects. In this step, a key part is to advect (animate) the particles by the simulated velocity fields. Figure 4.10 shows the particle animation, which is driven by the vector grids. A customized Python SOP operator is used for driving the particles. The workflow of this operator is:

1. Import the vector grids sequence into Houdini frame by frame
2. Get the linear interpolated velocity vector at current particle position
3. Move the particle by this vector
4. Repeat step 2, 3 to move all particles

After the particles are advected by the velocity fields, a Lattice SOP reshapes the “mushroom” of explosion volume. Figure 4.11 shows the comparison. This pipeline to animate particles is a critical component of the final curling and buoyant volume effects.

4.6 Particle based pyroclastic volume

Pyroclastic volumes are displaced spherical volumes, using noise displacement (Perlin Noise Fractal Sum) on the sphere surface [8]. Following is the pyroclast formulas (Figure 4.12):

\[
\vec{Y}(\vec{p}) = \frac{\vec{p} - \vec{c}}{|\vec{p} - \vec{c}|}
\]

\[
N(\vec{x}) = |FS(PN(\vec{x}))|^\gamma
\]

\[
Pyro.density(\vec{p}) = R + N(\vec{Y}(\vec{p})) - |\vec{p} - \vec{c}|
\]
Figure 4.12: Pyroclastic volume definition

Figure 4.13: Pyroclastic sphere with different parameters
Figure 4.14: Particle based pyroclastic volume

N is the displacement on the sphere surface; FS is fractal sum, PN is Perlin Noise value based on a position. Figure 4.13 shows some pyroclastic spheres with the various parameters, which were built using the above formulas. The particle based pyroclastic volume is composed of many pyroclastic volume spheres. Each pyroclastic unit uses its corresponding particle position as its center. This kind of volume technique is very powerful. It can create a continuously changing volume, since each sphere can have independently evolving parameters, such as radius(R), color, opacity and Perlin Noise. In this project, the smoke effect is mainly created by the particle based pyroclastic volumes.

There is a tricky issue about sampling the density and color of the pyroclastic volumes in overlapping area during Ray Marching process. For getting a smooth volume shape, the rendering program processes the density and color as following formulas:

\[ density = \max(\text{den}_1, \text{den}_2, \text{den}_3, \text{den}_n) \]
\[
weight_i = \frac{den_i}{\sum_{i=1}^{n} den_i}
\]
\[
color = \sum_{i=1}^{n} col_i weight_i
\]

Where \(den\) is the density value of each pyroclastic unit at the sampling point; \(col\) is the color of each sphere; \(n\) is the number of volume unit at sampling point. Following is the basic procedure to generate particle for the pyroclastic volume:

1. Use Houdini particle operators to generate the particles, then create necessary attributes through Houdini Create Attribute operator, such as pscale, noise parameters, colors and so on.

2. Use the simulated velocity fields to drive the particles to move.

3. Set other SOPs, like lattice and VOPs, to modify the final particles in order get the ”mushroom” shape.

4. Use Attribute operator to remove the unused particle attributes.

5. Export the particles into GEO files frame by frame, which are ASCII format.

6. Use a custom GEO reader to load the particle data and generate particle in VRP.

7. Generate pyroclastic volumes based on the imported particles and sample the density and color of the volume onto sparse grids.

The custom GEO reader plays an important role in generating the pyroclastic volume. This reader is responsible for effectively decode the GEO files. A noticeable advantage of this volume technique is that the volume animation can be intuitively created in Houdini’s particle system. The high controllability and flexibility of this volume technique contribute to achieve the artistic goals.
4.7 Match VRP camera with Houdini’s

Matching the VRP camera with Houdini’s is one of basic necessities for this production pipeline. A visually interactive pipeline is very important for the artistic jobs, especially for camera layout. Artists can easily set the camera angle and scene layout without rendering test images. For matching the two cameras, VRP has to set the camera’s field of view (FOV) as Houdini does. Camera FOV in Houdini is defined by focal length and aperture, like Figure 4.15 illustrates. The camera FOV in VRP is defined by the same way. In addition, for conveniently setting camera movements and Ray Marching, the camera in VRP was equipped with three auxiliary parameters: camera aim, near plane and far plane. The camera aim is designed for setting the camera direction; the near and far plane is used to define the space of Ray Marching. Figure 4.16 illustrates the camera relation between the two programs: VRP and Houdini. The camera local coordinates are computed by the following formulas:

\[
\begin{align*}
  \text{cam.vz} &= (\text{cam.aim} - \text{cam.pos}).\text{normalized()} \\
  \text{cam.vx} &= (\text{cam.vz} \times \text{cam.up}).\text{normalized()} \\
  \text{cam.vy} &= (\text{cam.vx} \times \text{cam.vz}).\text{normalized()}
\end{align*}
\]

4.8 Post-production

Post-production includes compositing and editing. Compositing in Nuke focuses on color correction and a glow effect. The color correction process adjusts the image contrast and color in order to make the explosion FX more real visually. The glow effect that is hard to be generated in the rendering stage improved the fireball
Figure 4.15: Houdini camera FOV

Figure 4.16: Camera definition in volume rendering program
Figure 4.17: The comparison between the source image and the compositing image elements. Figure 4.17 shows the great visual difference between the original image and the image with post-processing in Nuke. The creative edit in Premiere includes changing frame rate, add sound track and encode final video. The process to change frame rate is a useful tip to improve the FX timing in order to creates the ease-in and ease-out rhythm.
Chapter 5

Results

This project brings two major results: a custom volume FX production pipeline and a explosion FX animation created in this pipeline.

5.1 Volume FX production pipeline

The volume production pipeline is a major goal of this project. The pipeline includes three types of asset: C++ code, Python rendering scripts and custom Houdini operators. The C++ code plays an important role in the fluid simulation and the volume FX rendering with various algorithms and implementations, ad is the core of this project and the pipeline. The Python scripts support multiple tasks. The rendering program is configured and executed in Python environment. Many supplementary tools were coded in Python, to submit rendering jobs to the Palmetto cluster, to manager the rendering jobs and files, and to generate the SWIG interface files etc. The custom Houdini operators make the FX production more flexible and controllable. The application of Houdini in this project is also a critical component of this production pipeline. The explosion effect benefits from the custom Houdini op-
Figure 5.1: The explosion FX
video link[ https://vimeo.com/45293065 ]
erators, which improved the motion, shape and color. In short, this controllable and flexible FX pipeline satisfied the artistic requirements of the explosion FX production.

5.2 Explosion FX video

The explosion animation (Figure 5.1) is the important video output of this project. In artistic terms, it mimics the major features of real explosions, like curling turbulence motion, color changing and shape transformation. The video accompanied by the sound track conveys the explosion to audience visually and audibly. In technical terms, the video shows the fluid simulation and custom volume effect. Finally, the explosion video exhibits what the custom FX pipeline can do in the real production.
Chapter 6

Conclusions and Discussion

The most valuable achievement of this project is the custom volume FX production pipeline. The pipeline combines the custom program with commercial software Houdini together. The pipeline is very flexible, effective. It provides the ability to generate the volume FX by the complete custom algorithm and implementations, which help the artists to meet some special artistic goals in real production work. This pipeline can also be extended to other commercial software. By now, the project only achieved the main goals in artistic and technical aspects. If I have more time, I would like to do the further improvements in arts and technologies.

6.1 Improvement

Artistically, the explosion FX could be refined in its fire pattern on the fireballs elements, with more bumpy detail on the volume surface, so that the explosion will more closely follow the real world. The explosion motion could be improved too. For example, the overturning smoke can be sped up at that moment when the smoke is going to the center from the surface round the "mushroom" shape.
From a technical perspective, the pipeline still has much to improve. The most useful item would be to make the rendering process automatic and procedural. Currently the rendering configuration is a little complicated. It totally depends upon the manually written Python scripts. So a custom scene graph file can make the pipeline better.

Because the rendering is based on Python scripts, the scene graph file actually is a bunch of Python scripts, which are used to set the rendering parameters and call the C++ library to render. The custom scene graph file is similar to Nuke scene file, which uses xml codes to build Nuke compositing networks, or Maya ma scene file, which is an ASCII format file, using MEL code to build the Maya scene graph. Houdini operators could be built to execute the C++ library. For example, any C++ class could have a Houdini operator to represent it, and the Houdini operator generates the relevant Python scripts that are used to set the final rendering parameters and configuration. Then create another container SOP, which collects all of the relevant volume operators and writes out the scene graph file according to the Houdini network. Finally, a ROP in Houdini could load the scene graph file and submit the rendering commands. Depending upon the scene graph design, the user or FX artist would interactively render the volume FX in Houdini. This would definitely boost the production efficiency.

6.2 Future

The working mode of this pipeline, combining a custom program and commercial software, is very flexible, and fits into production environments. The volume production pipeline could extend into Autodesk Maya or Blender, both of which are equipped with a Python API. So it has high potential for real industrial production.
Appendices
Appendix A  Python Code to convert grids to Houdini points

1

# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
# VolRender Grids Node for Houdini
# JHY, Clemson U
# 2012−05−10
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

import hou  # load HOM API
print('load_vol_grid.py!')

def main():
    # set global variables for current node
    sm = hou.session  # global grids variables in Houdini session module
    vr = sm.vr  # the Python module of the volume rendering program
    cn = hou.pwd()  # current SOP node
    geo = cn.geometry()  # current geometry

    # GUI
    fn = cn.parm('file').eval()
    ty = cn.parm('type').eval()
    ifgenpts = cn.parm('if_genPts').eval()
    mod = cn.parm('mod').eval()
    thresh = cn.parm('threshold').eval()

    # grid type
    v_g = ''
    st = False
    if ty==0 :
        v_g = sm.floatGrids
        if mod=='binary' :
            st = v_g.readBinary(fn)
        else :
            st = v_g.read(fn)
        geo.addAttrib(hou.attribType_Point, 'Den', 0.0, False, True)
if ty==1:
    vg = sm.colorGrids
    if mod=='binary':
        st = vg.readBinary(fn)
    else:
        st = vg.read(fn)
    geo.addAttrib(hou.attribType.Point, 'Cd', (0.0, 0.0, 0.0), True, True)
if ty==2:
    vg = sm.vectorGrids
    if mod=='binary':
        st = vg.readBinary(fn)
    else:
        st = vg.read(fn)
    geo.addAttrib(hou.attribType.Point, 'v', (0.0, 0.0, 0.0), True, True)

# create point attributes to store the grid indices
geo.addAttrib(hou.attribType.Point, 'i', -1, False, True)
geo.addAttrib(hou.attribType.Point, 'j', -1, False, True)
geo.addAttrib(hou.attribType.Point, 'k', -1, False, True)

# retrieve the current grid information and display them on GUI
on3 = (1,1,1)
off3 = (0,0,0)
on4 = (1,1,1,1)
off4 = (0,0,0,0)

        cn.parmTuple('center').lock(off3)
        center = vg.getCenter()
        cn.parmTuple('center').set((center.X(), center.Y(), center.Z()))
        cn.parmTuple('center').lock(on3)

cn.parmTuple('size').lock(off3)
        cn.parmTuple('size').set((vg.getSX(), vg.getSY(), vg.getSZ()))
        cn.parmTuple('size').lock(on3)

cn.parmTuple('PBN').lock(off3)
        cn.parmTuple('PBN').set((vg.get_PBNx(), vg.get_PBNy(), vg.get_PBNz()))
        cn.parmTuple('PBN').lock(on3)

cn.parmTuple('space').lock(off3)
cn.parmTuple('space').set((v.g.getDx(), v.g.getDy(), v.g.getDz()))

if ty==0:
    cn.parm('outvalue').set(v.g.getOutvalue())
    stp = False
    if vv>thresh :
        stp = True
if ty==1:
    if vv.X()>thresh or vv.Y()>thresh or vv.Z()>thresh or vv.W()>thresh :
        stp = True
if ty==2:
    if vv.X()>thresh or vv.Y()>thresh or vv.Z()>thresh :
        stp = True

# create Houdini geometry points
if st:
    # get the point number in x, y and z axes
    nx = v.g.getNX()
    ny = v.g.getNY()
    nz = v.g.getNZ()
    for k in range(nz):
        for j in range(ny):
            for i in range(nx):
                vv = v.g.getValue(i,j,k)

                stp = False
                if ty==0:
                    if vv>thresh :
                        stp = True
                if ty==1:
                    if vv.X()>thresh or vv.Y()>thresh or vv.Z()>thresh or vv.W()>thresh :
                        stp = True
                if ty==2:
                    if vv.X()>thresh or vv.Y()>thresh or vv.Z()>thresh :
                        stp = True
if stp and igenpts:
    pos = v_g.getPos(i, j, k)  # get the 3D coordinate of current grid point in world space
    pos = (pos.X(), pos.Y(), pos.Z())
    pt = geo.createPoint()  # create geometry point with HOM API
    pt.setPosition(pos)  # set the coordinate for the current geometry point
    # set grid indices of current point
    pt.setAttribValue('i', i)
    pt.setAttribValue('j', j)
    pt.setAttribValue('k', k)

    # set grid value onto current point
    if ty==0:
        pt.setAttribValue('Den', vv)
    if ty==1:
    if ty==2:
        pt.setAttribValue('v', (vv.X(), vv.Y(), vv.Z()))
Bibliography


