12-2012

A Node Based Volume Modeling and Rendering Toolkit for Python

Robert Kern
Clemson University, rjkern@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/1521

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.
A Node Based Volume Modeling and Rendering Toolkit for Python

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
Robert Justin Kern
December 2012

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

This thesis covers the implementation and use of a node based volume modeling and rendering toolkit for python. The first part of the thesis covers the code structure of the toolkit, the implementation of the nodes, and how they are evaluated. Operator graphs are built from nodes to create and shape volumes. During volume rendering these graphs are evaluated for the density and color values. The second part of the thesis covers how the toolkit was used to create and render volumetric effects by emulating an effect seen in a movie. The animation of the effect, the use of the toolkit and compositing the effect are all discussed.
Table of Contents

Title Page ................................................................. i
Abstract ................................................................. ii
List of Figures ......................................................... iv
1 Overview ............................................................... 1
  1.1 The Code ......................................................... 1
  1.2 The Effect ....................................................... 2
2 Code Structure ....................................................... 4
  2.1 Fields ............................................................. 4
  2.2 The eval() Method ............................................... 6
  2.3 Primitives ....................................................... 7
  2.4 Operator Nodes ................................................. 10
  2.5 Grids ............................................................ 13
  2.6 Sparse Grids .................................................... 14
  2.7 Deep Shadow Maps ............................................. 14
  2.8 Rendering ....................................................... 15
3 Creating the Effect .................................................. 18
  3.1 Scene Animation ................................................ 18
  3.2 Particle Animation .............................................. 19
  3.3 Exporting Particle Data ........................................ 22
  3.4 Volume Generation .............................................. 24
  3.5 Compositing Effect Into A Background ....................... 29
4 Conclusion ............................................................ 32
  4.1 Possible Improvements .......................................... 32
Appendices ............................................................... 33
  A Script For Reading In Grids And Rendering The Effect .......... 34
  B Toolkit Scalar Field Node Definitions .......................... 37
Bibliography ............................................................. 41
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Frame from <em>Wrath of the Titans</em> trailer that inspired the effect.[1]</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Corresponding frame generated by the toolkit.</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Building an operator graph in python using the toolkit.</td>
<td>5</td>
</tr>
<tr>
<td>2.2 The operator graph built by the above python script.</td>
<td>5</td>
</tr>
<tr>
<td>2.3 The Volume created by the above python script.</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Some of the volume rendered scalar primitives. Left to right, top to bottom. Sphere, Torus, cube, cone.</td>
<td>8</td>
</tr>
<tr>
<td>2.5 The Planckian locus in the CIE color space. [5]</td>
<td>9</td>
</tr>
<tr>
<td>2.6 A pyroclast with the firecolor node applied.</td>
<td>9</td>
</tr>
<tr>
<td>2.7 From left to right, two spheres that have a union node, intersection node and cutout node.</td>
<td>11</td>
</tr>
<tr>
<td>2.8 A pyroclast after multiple rounds of advection.</td>
<td>12</td>
</tr>
<tr>
<td>3.1 The animation of the shot in Houdini.</td>
<td>19</td>
</tr>
<tr>
<td>3.2 The POP network of the impact particles.</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Operator graph generated for each pyroclast in the fire generation script.</td>
<td>25</td>
</tr>
<tr>
<td>3.4 An undesired halo can be seen around the fire.</td>
<td>26</td>
</tr>
<tr>
<td>3.5 The floor node can be used to get rid of the halo around the fire.</td>
<td>26</td>
</tr>
<tr>
<td>3.6 Operator graph generated for the fire in the final render script.</td>
<td>27</td>
</tr>
<tr>
<td>3.7 Operator graph generated for the scatter volume in the final render script.</td>
<td>27</td>
</tr>
<tr>
<td>3.8 Rendered frame from the effect.</td>
<td>28</td>
</tr>
<tr>
<td>3.9 Alpha image of the same frame for use in compositing.</td>
<td>28</td>
</tr>
<tr>
<td>3.10 The Nuke tree of the composite.</td>
<td>29</td>
</tr>
<tr>
<td>3.11 The background that the effect is composited into.</td>
<td>30</td>
</tr>
<tr>
<td>3.12 The background after it has been masked for the effect.</td>
<td>30</td>
</tr>
<tr>
<td>3.13 The final composite of the effect.</td>
<td>31</td>
</tr>
</tbody>
</table>
Chapter 1

Overview

This project has two purposes, (1) to develop a node based set of volume modeling and volume rendering tools for python and (2) use those tools to create and render a volumetric effect. The toolkit is written in C++ and the package SWIG is used to generate the python code. The effect is animated with POPs in Houdini from Side Effects Software and then volumetrically modeled and rendered with python scripts using the toolkit.

1.1 The Code

There are two main types of classes at work in the software. The 'Volume' classes do all of the heavy lifting while 'Field' classes are used in python scripting. Every node in the toolkit is a subclass of the 'Volume' class and performs the operation of that node. There are nodes that handle scalar operations, vector operations and color operations as well as primitives for each of these types.

Python has an aggressive garbage collection scheme. This can create problems during scripting because of the way volumes are modeled in the toolkit. Python’s garbage collection can, in some cases, deallocate nodes in the operator graph causing the script to crash. To prevent this, the 'Field' class wraps the volume object using the C++ standard library shared_ptr class. The shared pointers help prevent premature deletion of volume objects during garbage collection in python. A set of C++ functions with Swig based python bindings are used to create and manipulate fields. These python commands are what are used in scripting.
1.2 The Effect

To exercise and illustrate how the toolkit works, a volumetric effect shot has been constructed. The effect created was inspired by an effect seen in the trailer for the movie *Wrath of the Titans* seen in Figure 1.1. [1] A corresponding frame from the generated effect can be seen in figure 1.2. While the shot is animated and framed in Houdini, the particle data for the volumetric fields is exported from Houdini to a set of files on a per frame basis. Python scripts use the toolkit to read this data, generate the volumes and render out the images. These images were composited into the rest of the shot. Because of the large amount of time and memory it takes to generate and render these volumes, the scripts are run on the Palmetto Linux Cluster at Clemson University. The goal was to create something that read as a flaming meteor flying through the sky leaving a smoke trail behind it and was visually appealing.

Figure 1.1: Frame from *Wrath of the Titans* trailer that inspired the effect.[1]
Figure 1.2: Corresponding frame generated by the toolkit.
Chapter 2

Code Structure

The majority of the code for the toolkit is built on 'Volume' classes and 'Field' classes. Every node has a corresponding class that is derived from the Volume class. These classes contain and perform all of the algorithms and operations for that node. The Volume classes for each node also stores any data that is needed to perform an operation. The eval() method in each volume class is responsible for the node evaluation. The Field classes are derived from the shared_ptr class in the C++ standard library. The fields act as reference-counted pointers for Volume objects. The three types of volumes and fields are float (which will be referred to as a scalar) vector and color.

2.1 Fields

The Field classes are what are used for scripting in Python. There are a set of functions within the toolkit for creating fields and building operator graphs with them. There are three types of fields that can be made. ScalarFields will return a floating point value when evaluated. VectorFields will return vectors. ColorFields will return red, green, blue, and alpha color values. As an example, calling the function 'sphere()' returns an instance of the ScalarField class holding a SphereVolume. The function 'translate()' creates a ScalarField that is a translation of the ScalarField that is passed in as an argument. The 'union()' function creates a ScalarField that is a combination of the two ScalarFields given as arguments. Figure 2.1 shows an example of building an operator graph in python using the toolkit.

The first line creates a ScalarField that contains a SphereVolume with a radius of 1 and
Figure 2.1: Building an operator graph in python using the toolkit.

assigns it to the python variable ‘v’. The next line creates a ScalarField that contains a TranslateVolume, which in turn takes the ScalarField ‘v’ and stores it within the TranslateVolume class. Because the ScalarField is stored in the TranslateVolume, the new ScalarField can be stored back in ‘v’ without losing the SphereVolume that was originally assigned to ‘v’. The third line shows that creating new nodes in the graph can be nested. In this line, a new sphere is created, translated by (-.5,0,0), unioned with the ScalarField from line 2 and then assigned back to ‘v’. Without the reference counting done by the shared_ptr class, python garbage collection would deallocate the SphereVolume as well as the TranslateVolume that are created in line three. Figure 2.2 is a representation of the operator graph and figure 2.3 shows the rendered volume created by the operator graph.

Figure 2.2: The operator graph built by the above python script.
2.2 The eval() Method

Every node in the toolkit has an eval() method available in the underlying volume object. This is used to find the output of the node at any position $\mathbf{x}$. The eval function is given $\mathbf{x}$ as its only argument and has a return type that matches what type of field the node represents. A ScalarField will return a float value, a VectorField will return a vector value and a ColorField will return a color value. This function is part of what makes the operator graph structure work. Most of the nodes in the toolkit either manipulate the output of its input node or manipulate the sample point $\mathbf{x}$ in some way.

As an example, here is how the operator graph made by the python script in figure 2.1 would evaluate when sampled at $\mathbf{x}$. The eval() function of the last node in the graph, the union node, is called and $\mathbf{x}$ is passed in as the argument. The union node’s eval() function is defined as $f_{\text{union}}(\mathbf{x}) = \max(f(\mathbf{x}), g(\mathbf{x}))$ where $f(\mathbf{x})$ and $g(\mathbf{x})$ are two scalar field inputs. In this case, they are translate nodes. The vector $\mathbf{x}$ gets passed on to the eval function of the translate nodes. The translate node’s eval() function is defined as $f_{\text{translate}}(\mathbf{x}) = f(\mathbf{x} - \mathbf{v})$. In this instance $f(\mathbf{x})$ is once again a scalar field input to the node. The translate nodes manipulate the sample point by subtracting the vector $\mathbf{t}$ from it before passing it on to the eval() function of the sphere node. The sphere node’s eval() function is defined as $f_{\text{sphere}}(\mathbf{x}) = R - |\mathbf{x}|$. This value is then returned to the translate nodes, then back to the union node. Once the second scalar field has evaluated, the union
node returns the maximum of the two values.

### 2.3 Primitives

Some of the nodes in the toolkit are leaves in the operator graphs. They are the most basic fields that can be created and so that makes them the primitives of the toolkit. For scalar volumes, there are a set of nodes that implicitly define a volumetric shape. There are nodes that define a sphere, a cube, a cone, a torus and more. Some of these primitives can be seen in figure 2.4. When the node’s eval function is called by the renderer, it is given a sample position, \( \mathbf{x} \). That sample position is used in the implicit function to determine the value of the volume at \( \mathbf{x} \). For example, a sphere is implicitly defined as

\[
f(\mathbf{x}) = R - |\mathbf{x}|
\]

where \( R \) is the radius of the sphere. A cone with the length "\( \text{height} \)" and angular width \( \theta \) is implicitly defined as

\[
f(\mathbf{x}) = \begin{cases} 
0 & \mathbf{x} = \mathbf{x}_0 \\
\text{height} - (\mathbf{x} - \mathbf{x}_0) \cdot \hat{n} & (\mathbf{x} - \mathbf{x}_0) \cdot \hat{n} > \text{height} \\
(\mathbf{x} - \mathbf{x}_0) \cdot \hat{n} & (\mathbf{x} - \mathbf{x}_0) \cdot \hat{n} < 0 \\
\theta - \cos^{-1} \left( \frac{(\mathbf{x} - \mathbf{x}_0) \cdot \hat{n}}{|\mathbf{x} - \mathbf{x}_0|} \right) & \text{otherwise}
\end{cases}
\]

For the vector volume classes, there is an identity node that when sampled at \( \mathbf{x} \), returns \( \mathbf{x} \). Another node creates a field \( U(\mathbf{x}) = \nabla \times FS(PN) \) where \( FS(PN) \) is a vector field made of perlin noise. This node takes in three float values, \( dx, dy \) and \( dz \) that are used to control the level of discretization in finding the curl of the perlin noise vector field. The `Noise_t` class that’s part of the tool kit holds all of the noise parameters for the perlin noise function. All nodes in the toolkit that use perlin noise take a `Noise_t` object as an argument to control the noise. In this case, the `Noise_t`
object shapes the VectorField. Because this field is the curl of perlin noise, it is also divergence free. This property makes it an ideal vector field to use for advecting volumes.

The firecolor node takes in a scalar field, $f(x)$, and a temperature value in kelvin, $k$. When sampled the value of $f(x)k$ is mapped to the Planckian locus in the CIE color space [4] and can be seen in figure 2.5. If the temperature was not multiplied by the value returned by the scalar field, the whole color field would be uniform. This helps add variety to the ColorField making it more visually appealing rather than a single, less interesting color. An example of what the effect of the fire node looks like can be seen in figure 2.6.

All three types of volumes, scalar, vector and color, have a constant node. This node always returns a user defined value regardless of the sample position.

Pyroclasts are a special type of primitive in the toolkit. A pyroclast is a sphere that has perlin noise displacing its surface. As its arguments, the pyroclast node takes in a radius, a value called gamma, an amplitude value and a Noise object that carries all of the parameters for the perlin noise function. Spheres are easiest to add this type of noise to since the normal at any given point on the surface is simple to find. Pyroclasts are defined the same as a sphere with an extra
Figure 2.5: The Planckian locus in the CIE color space. [5]

Figure 2.6: A pyroclast with the firecolor node applied.
term \( D(\mathbf{x}) \) added.

\[ D(\mathbf{x}) = \text{amplitude}|PN(\frac{\mathbf{x}}{|\mathbf{x}|})|^\gamma \quad (2.3) \]

\[ f_{\text{pyroclast}}(\mathbf{x}) = R - |\mathbf{x}| + D(\mathbf{x}) \quad (2.4) \]

### 2.4 Operator Nodes

Operator nodes are also derived from the Volume class. All of the operators take in a Field class as part of their input. Some of them take in multiple Fields. The operator nodes perform operations on the Fields returning a new Field. This creates a graph of these operator nodes. During rendering, the node at the end of the graph is evaluated by the volume renderer at each ray march step. As the graph is traversed, either the sample point or the value of the field is manipulated by the nodes to achieve the final result.

One set of operator nodes act as 3D transformations. The translate node takes a Field, \( f \), and a vector \( \mathbf{u} \). The output of this node is a Field that has been translated by \( \mathbf{u} \). The scale node takes a Field, \( f \) and uniformly scales it by some value \( L \). The rotate node rotates a Field by an angle \( \theta \) around an axis \( \hat{n} \).

Another set of operator nodes made specifically for scalar fields allow scalar fields to be used to construct new volumes. The union node combines two scalar fields into one using constructive solid geometry (CSG) operations [2]. Given scalar fields \( f(\mathbf{x}) \) and \( g(\mathbf{x}) \) the union of these two fields can be defined as

\[ f_{\text{union}}(\mathbf{x}) = \max(f(\mathbf{x}), g(\mathbf{x})) \quad (2.5) \]

The intersection node results in a scalar field that is positive only in the area that both of the input scalar fields simultaneously occupy. The intersection of two scalar fields can be defined as
Lastly, the cutout node removes the second scalar field from the first one and can be defined as

\[ f_{\text{cutout}}(x) = \min(f(x), -g(x)) \]  

(2.7)

There are two nodes available that can advect a volume. The advect node takes a ScalarField, a VectorField and a time step \( \Delta t \). The ScalarField is advected using semi-lagrangian advection.

\[ f_{\text{advect}}(x) = f(x - U(x)\Delta t) \]  

(2.8)

The VectorField, \( U(x) \), is used as a velocity field and \( \Delta t \) is used as the time step for advection. An example of a volume that has had the advect node applied to it can be seen in figure 2.8. Another method for advecting volumes in the toolkit is a combination of a function named selma, which stands for semi-lagrangian mapping, and the warp node. Selma as it is implemented here is based on the implementation of selma from the paper *Resolution Independent Volumes* [3]. The selma function takes a VectorField, a time step \( \Delta t \), and a grid as input. The VectorField serves as the velocity field for advection. The selma function finds how much space will be displaced by semi-lagrangian advection and stores this displacement in the grid. Selma can also store multiple rounds of advection on the same displacement grid. Selma iterates through every point in the displacement
grid and updates it as follows.

\[
grid_{ijk} = grid_{ijk} - U(x + grid_{ijk})\Delta t
\]  

(2.9)

Here \(grid_{ijk}\) stands for the value held by each grid point in the displacement grid. The velocity field is represented by \(U\).

![Figure 2.8: A pyroclast after multiple rounds of advection.](image)

The warp node takes in a ScalarField and a VectorField. The output of the warp node is the ScalarField displaced by the VectorField.

\[
f_{\text{warp}}(x) = f(x + U(x))
\]  

(2.10)

One of the advantages to using selma is that after advecting, only the warp node is added to the operator graph. Trying to use several rounds of advection using several advect nodes can cause significant increases in render time.

Some other useful operator nodes include mask, clamp and floor. One use for these is to eliminate areas of negative density which can cause problems during rendering.
\[ f_{\text{mask}}(x) = \begin{cases} 
0 & f(x) \leq 0 \\
1 & \text{otherwise}
\end{cases} \quad (2.11) \]

\[ f_{\text{clamp}}(x) = \begin{cases} 
\minValue & f(x) \leq \minValue \\
\maxValue & f(x) \geq \maxValue \\
f(x) & \text{otherwise}
\end{cases} \quad (2.12) \]

\[ f_{\text{floor}}(x) = \begin{cases} 
0 & f(x) < \minValue \\
f(x) & \text{otherwise}
\end{cases} \quad (2.13) \]

### 2.5 Grids

Whenever a volume or field is sampled, the entire operator graph must be evaluated to find the result of that sample point. For small operator graphs this is not much of a problem. However, as the operator graph grows, so does the time it takes to evaluate the operator graph. If the operator graph is large enough, it may become impractical to use for rendering simply because it takes too much time. By sampling the graph to a grid, this problem can be alleviated. Once a graph has been sampled onto a grid, the grid can be given to the 'gridfield' node which wraps the grid into a Field. The eval() function of the gridfield node finds the nearest grid points surrounding the sample point and interpolates the values stored at those grid points to find the value for the sample point. This can be done much faster than evaluating an operator graph of hundreds of nodes at every sample point during ray marching. More operator nodes can be applied to the gridfield node allowing for further manipulation of the field.

Grids are not without their own problems. A grid must have enough grid points to capture all of the desired detail. For creating a volume over a sequence of frames, the grid must be consistent through the sequence. If the grid is not consistent between frames, a flickering effect may appear. If the effect that is being created covers a large area and fine detail is desired the grid is going to
require a large amount of memory to store that data. Also, while the grid may need to be large, for any given frame, the grid may be mostly empty. This is a huge waste of memory.

2.6 Sparse Grids

To address the previously mentioned memory waste, the toolkit makes use of sparse grids and doubly sparse grids to save memory. Sparse grids partition the grid points into blocks. The grid has a pointer for each block. For scalar grids, these are pointers to floats. Vector grids have pointers to vectors and color grids have pointers to ‘color’ objects. By default, each of these pointers point to NULL. When getting the value of a grid point on the grid, if the pointer to the block the grid point resides in points to null, a default value is returned. When a grid point is set to hold something other than the default value and the block it is in is pointing to NULL, then an array containing all the grid points in that block is allocated. The pointer for that block is then set to point to this array.

Doubly sparse grids take this same idea to another level. Doubly sparse grids partition the blocks of the sparse grid into blocks of their own. When a grid point is set to a value other than the default, the blocks in that partition of the doubly sparse grid are then allocated as well as the grid points within the specific block that the grid point is located.

2.7 Deep Shadow Maps

To render a lit volume, the renderer needs to know the transmittance, or the amount of light that reaches any sample point from any lights. To find that value, a ray march must be performed from the sample point to the light. Trying to do this for every sample point during the rendering process is completely impractical. Deep shadow maps are a solution to this problem. Traditionally in computer graphics, shadow maps are responsible for shadows and nothing more. However, in volume rendering, deep shadow maps are responsible for not just shadows, but lighting as a whole. Without a shadow map, a volume is treated as though it is emitting light. Only with a shadow map on the volume is it treated as though light is shining on it. The deep shadow map is a grid where each grid point stores how much accumulated density is between it and a light source. The ray march for finding this value updates as follows.
\[ \mathbf{x} = \mathbf{x} + \mathbf{ray_{light}} \Delta \mathbf{x} \]

\[ \text{grid}_{ijk} = \text{grid}_{ijk} + f(\mathbf{x}) \Delta \mathbf{x} \]

Here, \( \mathbf{ray_{light}} \) is a unit vector from the grid point to the light source. This ray march continues until it goes outside the bounds of the grid, reaches the light source or goes past the light source. The final transmittance is calculated during rendering from this accumulated density. This is so that parameters related to the shadow map, such as the scatter coefficient, can be modified without the need to recreate the entire deep shadow map. To help speed up the creation of the deep shadow map, the density is sampled at that grid point. If \( f(\mathbf{x}) \leq 0 \), then there is no need to perform a ray march for that grid point. This is because the transmittance will eventually be multiplied by the density of the volume at \( \mathbf{x} \) and would end up becoming zero anyway.

### 2.8 Rendering

Before the render step, all the data for rendering is stored in the RenderData class. The RenderData class can store up to two separate volumes for rendering. One of these is an emissive volume, a volume that emits light, and the other one is a scatter volume, a volume that is lit by a light source. The RenderData class also stores the location in space of all the pixels of the image as well as all of the rays that go from the camera through the pixels. The render function takes in a RenderData class as input and using this, the renderer computes accordingly. For example, by default, the renderer performs a ray march from the camera to some distance. If a bounding box is found in the RenderData object, then the renderer will skip the ray march for any pixels who’s corresponding ray does not intersect the bounding box. If the ray does intersect the bounding box, the ray march will begin at the closest intersection and end at the farther intersection. The ray march also varies slightly depending on whether the RenderData class contains an emissive volume, a scatter volume or both. For rendering just an emissive volume, each step of the ray march proceeds as follows.
\[ x = x + ray_i \Delta x \]  
\[ \Delta T = e^{-f(x)k\Delta x} \]  
\[ c_{\text{Accum}} = c_{\text{Accum}} + T(1 - \Delta T)f(x)C(x) \]  
\[ T = T\Delta T \]  
\[ \alpha = \alpha + (1 - \Delta T)(1 - \alpha) \]

This ray march is based on the ray march found in *Volumetric Methods in Visual Effects*.\[6\] To render scatter volumes, lighting from deep shadow maps must be taken into account. To do this, a loop goes over each shadow map and updates a color value, \( c \).

\[ c = c + e^{k_{\text{DSM}}DSM_{\text{color}}}(x) \]  
\[ (2.19) \]

For this, \( k_{\text{DSM}} \) is the shadow map’s scatter coefficient and \( DSM_{\text{color}} \) is the color of the light that corresponds to the shadow map. This term is then worked into the color accumulation of the ray march. Equation 2.16 becomes

\[ c_{\text{Accum}} = c_{\text{Accum}} + T(1 - \Delta T)f(x)C(x)c \]  
\[ (2.20) \]

When rendering both an emissive volume and a scatter volume simultaneously, \( c \) is calculated the same way. The changes come in equations 2.15 and 2.16 of the ray march.

\[ \Delta T = e^{-(f_{\text{scatter}}(x) + f_{\text{emissive}}(x))k\Delta x} \]  
\[ c_{\text{Accum}} = c_{\text{Accum}} + T(1 - \Delta T) \frac{f_{\text{scatter}}(x)C_{\text{scatter}}(x)c + f_{\text{emissive}}(x)C_{\text{emissive}}(x)}{f_{\text{scatter}}(x) + f_{\text{emissive}}(x)} \]

At the end of the ray march for a pixel, that color value is stored within the RenderData class. These values can later be passed on to the image class using a few different functions to produce a few different types of images. The createImage function creates an image using the rgb channels. There are also functions for generating images of the red channel, blue channel, green channel and the alpha channel. The image is then written to a file in OpenEXR format using the
Open Image IO library.
Chapter 3

Creating the Effect

To illustrate how the toolkit works, it was used to create a volumetric effect. This effect had to be visually and artistically appealing or the toolkit could not prove its usefulness. The parameters that controlled the shape of the pyroclasts had to be animated so that for each of the volumes, they moved in a way that appeared believable. All of the smoke volumes started out sharp and noisy and then became softer and more rounded to appear as though they were dissipating in the air. The fire had to look as though it was wildly burning and had to blend into the smoke. The smoke thrown up by the impact of the meteor needed to be thrown violently into the air to convey the strength of the meteor’s impact into the ground. The environment and the camera animation needed to help convey the scale and sense of space around the meteor to pull the whole effect together and make it more readable.

3.1 Scene Animation

While the software was used to generate and render the volumes, the animation of the effect was actually done with the 3D package Houdini from Side Effects software. Houdini was used since it is the most often used in the visual effects industry for creating special effects. A sphere animated to follow a curve served as the trajectory of the meteor. The sphere also served as the emitter for fire and smoke particles that were supposed to be part of the meteor. A flattened torus at the point of the meteor’s impact served as the emitter for dirt and dust particles thrown up by the impact of the meteor. A camera was animated to follow the meteor from its launch point to its impact point.
3.2 Particle Animation

There were three particle systems in the shot. There was one for the fire produced by the meteor, one for the smoke and one for the dirt and dust thrown in the air by the meteor’s impact with the ground. The particles for each of these were emitted by some piece of geometry. In the case of the fire and smoke it was a sphere, and a torus in the case of the impact volume. All three POP networks start with a 'Source' node. This node turns the vertices of the geometry into points where particles can be emitted. This node also controls how the particles are emitted. The birth rate for the smoke and fire were both set to 50 particles per second. In the fire POP network, this immediately dropped to 0 once the meteor impacted the ground. For the impact source node, the birth rate was 0 until the meteor impacted the ground. The birthrate then briefly jumped up to 500 particles per second and then ramped back down to 0 over the course of 9 frames. In all of the Source nodes, the 'Accurate Births' box was checked. This interpolates the correct sub-frame position for the emitted particles. The initial velocities are also set by the source node. For the smoke and fire, the initial velocities were set to (0, 0, 0) with a variance of (1, 1, 1) to give them just
a little bit of motion. For the impact particles the initial velocities started at (0, 209.861, 0) and ramped down to (0, 30.078, 0) over the course of 7 frames. This launched the first particles high in the air quickly and the slowing initial velocity helped the impact particles to spread out so that the volume they generated would look continuous. They also had a variance of (35.515, 20, 36.584) so that they spread out in the x and z direction as they go up.

In the POP graphs for each of these particle systems, there are several 'Add Attribute' nodes that were used to control parameters for generating the volumes. The attributes that were added to the particles are amplitude, color, fjump, gamma, octaves, opacity, radius, roughness, wavelength. While some of these parameters remained constant throughout the duration of the shot, most were animated with python code or with expressions within Houdini. None of the particles for the three systems die during the duration of the shot, but they do not remain visible during the entire time. The 'Add Attribute' node called 'vislife' created a variable for the POP network by the same name. The value of this variable for the smoke trail POP was controlled by the following python code executed in the POP.

```python
normage = lvar('AGE')/ (3.75 + (.5 * hou.hmath.rand(lvar('ID'))))
if normage > 1:
    return 1
else:
    return normage
```

The script gave the 'vislife' variable a value from 0 to 1 which represented the age of the particle. The script takes the age, in seconds, that a particle has been alive and divides it by a lifespan. In this case, the lifespan is 3.75 seconds plus some variant amount. Houdini’s rand() function returns a value from -1 to 1. By giving the rand() function the id of the particle as the seed, some variation was created in the lifespan between particles. Also by using the ID of the particle, this lifespan value remained consistent from frame to frame for each particle. The value returned by rand() is multiplied by .5 and then added to the base lifespan to produce the final lifespan for that particle.

The 'vislife' variable created by this node was used to control most of the other parameters for volume generation.

Opacity is one of the attributes in the POP networks that was controlled by a python script and used the vislife variable to control its value. The following script controlled the opacity of the smoke trail.

```python
```
if lvar('VISLIFE') < .1:
    return 0
eelif lvar('VISLIFE') < .15:
    return 1 - ((.15 - lvar('VISLIFE')) / .05)
else:
    return (1 - lvar('VISLIFE')) / .85

Since the beginning of the trail is where the fire is and the smoke should only appear after the fire, if the 'vislife' variable is less than .1, the opacity is 0. After that, the opacity quickly ramps up to 1 and then slowly ramps back down to 0. While opacity controls the visibility of the pyroclasts the other attributes control the shape of the pyroclast. For the smoke trail, these were animated using python scripts and expressions so that they go from a rough noisy look to a more smooth dissipated look. The octaves attribute controls how many rounds of fractal sums are performed on the perlin noise displacing the pyroclast. For the smoke trail, octaves was controlled with an expression and was based off of the value of vislife.

\[ 2.8 - (.3 \times VISLIFE) \]

As the value of octaves decreases, the pyroclast takes on a smoother appearance. A roughness variable that was also controlled with an expression and was based on the value of vislife. The following is the expression that controlled roughness.

\[ .5 + (.1 \times VISLIFE) \]

Despite the value of roughness increasing overtime, the desired look was still produced. The gamma attribute can be used to either flatten and round, or raise and sharpen, the peaks formed by the perlin noise. The desired look for the smoke trail was for the peaks to start flat and more rounded and then to expand out. The value of gamma was controlled by the following python script.

if lvar('VISLIFE') < .1:
    return .2
eelif lvar('VISLIFE') < .2:
    x = 1 - ((.2 - lvar('VISLIFE')) / .1)
    return .2 + (.8 * x)
eelse:
    return 1

The value of gamma does not start changing until vislife is greater than or equal to .1 which is when the pyroclast becomes visible. Once the pyroclast is visible, gamma goes from .2 to 1 and then stays
there. One of the side effects of a growing gamma value was that the pyroclast appeared to shrink. Since the desired look was to have the pyroclast grow and spread as though it was dissipating, the amplitude attribute was used to help minimize the shrinking due to the changing gamma. The amplitude was animating using the following python code.

```python
if lvar('VISLIFE') < .1:
    return .5
elif lvar('VISLIFE') <.4:
    x = 1-((.4 - lvar('VISLIFE'))/.3)
    return .5 + (.5 * x)
else:
    return 1
```

In addition to animating the amplitude of the noise, the overall radius of the pyroclast was animated. This was done with the following expression.

$$2.5+(1*VISLIFE)$$

One final attribute, wavelength, was also animated with an expression.

$$1.3 - (.7*VISLIFE)$$

The POP networks for the fire and the impact volumes have similar structures, differing primarily in the values used in the python code and expressions. The impact POP also has three additional nodes, a 'gravity' node, a 'drag' node and a 'noise' node. These nodes helped animate the movement of the particles themselves. The impact particles are launched quickly into the air, but the gravity node and drag node cause them to quickly slow down and begin to drift around slowly. The noise node helped add a little turbulence to the motion of the impact particles. These were not necessary on the smoke and fire particles since their movement upon their emission already had a desired look.

### 3.3 Exporting Particle Data

The particle and camera data was exported to a series of text files using a python script. The script loops over every frame in the shot and generates a data file for each frame. Also, there was a separate set of simulation files written for each of the three particle systems. The first line of the simulation file contains all of the necessary camera data. This includes the camera’s position,
direction vector, up vector, aspect ration, focal length and aperture. All the remaining lines contain information about each individual particle from one of the three particle systems. It has the values for each of the parameters added by the 'Add Attribute' nodes as well as the particle's position, velocity and age.
### 3.4 Volume Generation

The volumes for this effect were generated in three different scripts and then rendered in a fourth script. Separating the volume generation this way has a few advantages. For example, the volume that makes up the dirt thrown into the air by the meteor’s impact does not need nearly as large of a grid as the trail itself since it is isolated to a relatively small area. By generating them separately, a grid that is more appropriate for the volume can be used. Also if adjustments have to be made to just one of the three volumes, time is not wasted needlessly recreating the other two.

The first script generated the volume for all of the dust that is thrown up in the air upon the meteor’s impact. This was done by reading the particle data file for these particles for that frame. For every particle a pyroclast was generated using the noise parameters that were written to the file for that particle. A translate node was put on the pyroclast to place it at the same point in space as the particle. Multiple advect nodes were then added to generate some fine wispy detail to the structure. The scalar operator graph was then sampled to a doubly sparse grid as follows.

\[
\text{grid}_{ijk} = \max(\text{grid}_{ijk}, f(x_{ijk}))
\]  

Bounding boxes were used around each pyroclast to help speed up the time it takes to sample the pyroclast. Each pyroclast was not sampled at points outside of the bounding box. After sampling all of the pyroclasts onto the grid, the scalar grid was written to a file. The second script used the same procedure to generate the scalar grid for the smoke of the meteor trail. Since the smoke and impact volumes are constant colors in the final render, it was more efficient to create the color field using the constant and multiply nodes in the final render than to generate color grids for them. The volume for the fire was also generated in a similar manner, however the color was created through the use of the firecolor node which was written out to a grid. While sampling the fire volume to the grid, if \( f(x_{ijk}) > \text{grid}_{ijk} \) then the value stored on the doubly sparse color grid at \( \text{grid}_{ijk} \) was replaced by \( C(x_{ijk}) \).

The fourth and final script read the scalar grids for each of the three volumes as well as the color grid for the fire. The scalar grid and the color grid for the fire volume were given to a pair of gridfield nodes to turn them into fields. When the fire color was generated, not all of the
scalar field it is based on reached a high enough temperature to produce a color. Anywhere that does not reach at least 1667 kelvin does not produce a color. However these areas still render as opaque in the alpha image creating an undesired black halo in the final composite. To remedy this, before rendering, a floor node was attached to the fire scalar field with a minimum value of .95. This trimmed off the dark area. The resulting scalar field and its corresponding color field were given to the render data class to be rendered as an emissive volume. The scalar grids for the smoke and impact volumes were turned into fields using the gridfield node and then the two were combined to
make one volume using the union node. To generate their color fields, a constant color node was created and then the multiply node was used so that the color field was multiplied by the value of the scalar field.

Figure 3.4: An undesired halo can be seen around the fire.

Figure 3.5: The floor node can be used to get rid of the halo around the fire.

Since the combined smoke and impact volume is a scatter volume, deep shadow maps were generated for the lighting information. The scatter volume along with its color field and the shadow maps are stored within the RenderData class.
To get the correct camera data, one of the simulation files for the volumes was given to the readCameraDataFile() function. This just reads the first line of the simulation file which has all
of the necessary camera data including the position, direction, up vector and field of view of the camera. The camera was used by the RenderData class’s initializeRays() method, along with the width and height of the image. This method generates all of the rays that will be used for rendering the image. Once completed, the rendered color values for the pixels that were stored as an image file, using Open Image IO library, in OpenEXR format. Lastly, an alpha channel was written to a separate file for use in compositing.

Figure 3.8: Rendered frame from the effect.

Figure 3.9: Alpha image of the same frame for use in compositing.
3.5 Compositing Effect Into A Background

To make the effect easier to read and give it context, the effect was composited into a background. The environment was created and rendered in Maya. To get the camera of the effect and the camera of the background to match, the camera animation from Houdini was exported as an .fbx file. This file was read into a maya session containing modeled ground plane and a sky sphere. The ground plane and the sky sphere were rendered out of Maya separately. The compositing was done with Nuke. Nuke, like the toolkit and Houdini, works using a node based system. The Nuke node tree used to composite the effect is shown in Figure 3.10. The image sequences of the ground and sky are color corrected. The ground layer was put over the sky layer with a merge node set to 'over' mode and the opacity image sequence of the ground was also connected to the merge node as a mask. The result of the background can be seen in figure 3.11. The alpha channel was read and inverted, then merged with the background removing the area from the background where the
effect goes (figure 3.12). The rendered effect was brought in and attached to a color correct node.

Using the color correct node to alter the gamma and gain of the image, the colors in the rendered effect were brought out. This was added to the composite using a merge node set to ‘plus’ mode, as shown in figure 3.13. The sequence of composited images were written to disk as a set of images and in a video format.
Figure 3.13: The final composite of the effect.
Chapter 4

Conclusion

The toolkit provides a several nodes and functions that can be used and combined to create a variety of volumetric effects. Since the tool kit can be used in python, these effects can be created with simple scripts instead of writing and compiling whole C++ programs. The meteor effect created shows that the toolkit can be used for creating visually appealing effects. The meteor makes an arc as it speeds across the sky before making an explosive impact. It also makes use of both scatter volumes and emissive volumes and manages to blend them together quite well.

4.1 Possible Improvements

During rendering, each pixel’s value is determined through the ray march independent of any of the other pixels in the image. This makes it easily parallelizable. Currently, the toolkit does not render multiple pixels in parallel. Adding this feature could significantly reduce render times.

While the renderer does make use of bounding boxes, it could make greater use of them. Currently, the renderer just looks for a starting point and an ending point for the ray march using any bounding boxes found in the render data class. Skipping empty space between bounding boxes could help improve render times. Using BVH trees could also help speed up rendering several volumes.
Appendices
Appendix A  Script For Reading In Grids And Rendering

The Effect

#!/usr/bin/python

import os
import sys
import math
from volumes import *
from vrutils import *

cam = Camera()
rd = RenderData()

#Read in command line values
width = int(CmdLineFind("-width", 640))
height = int(CmdLineFind("-height", 360))
name = str(CmdLineFind("-name","/home/rjkern/projects/RJKERN_volrend/images/smoke"))
extention = str(CmdLineFind("-extension", "exr"))
rd.k = float(CmdLineFind("-scatter", 1.0))
rd.dsmk = float(CmdLineFind("-dsmScatter",50.0))
rd.deltaS = float(CmdLineFind("-deltaS", .05))
rd.maxS = float(CmdLineFind("-maxS", 10))
dsmres = float(CmdLineFind("-dsmres",10))
camfile = CmdLineFind("-camfile"," /home/rjkern/projects/RJKERN_volrend/simFiles/0003/particleDataSystem2")
impactgrid = CmdLineFind("-impactgrid","")
impaclcolor = CmdLineFind("-impactcolor","")

nx = int(CmdLineFind("-nx", 320))
ny = int(CmdLineFind("-ny", 460))
nz = int(CmdLineFind("-nz", 460))

inx = int(CmdLineFind("-inx", 300))
in = int(CmdLineFind("-iny", 300))
inz = int(CmdLineFind("-inz", 300))

dsmax = int(CmdLineFind("-dsmx", 320))
dsmy = int(CmdLineFind("-dsmy", 320))
dsmy = int(CmdLineFind("-dsmz", 320))

gridWidth = float(CmdLineFind("-gridwidth", 52.0))
gridHeight = float(CmdLineFind("-gridheight", 64.0))
gridDepth = float(CmdLineFind("-griddepth", 66.0))

igw = float(CmdLineFind("-igw", 20))
igh = float(CmdLineFind("-igh", 30))
igd = float(CmdLineFind("-igd", 23))

llcx = float(CmdLineFind("-llcx", -27.0))
llcy = float(CmdLineFind("-llcy", -14.0))
llcz = float(CmdLineFind("-llcz", -18))

llcxi = float(CmdLineFind("-llcxi", -10.0))
llcyi = float(CmdLineFind("-llcyi", -5.0))
llczi = float(CmdLineFind("-llczi", 372))

readSmoke = str(CmdLineFind("-readsmoke",""))
readFire = str(CmdLineFind("-readfire",""))

readSmokeColor = str(CmdLineFind("-readsmokecolor",""))
readFireColor = str(CmdLineFind("-readfirecolor",""))
readdsm1 = str(CmdLineFind("-readdsm1",""))
writedsm1 = str(CmdLineFind("-writedsm1",""))

readdsm2 = str(CmdLineFind("-readdsm2",""))
writedsm2 = str(CmdLineFind("-writedsm2",""))

print 'Image width: ' + str(width)
print 'Image height: ' + str(height)

print 'Grid nx: ' + str(nx)
print 'Grid ny: ' + str(ny)
print 'Grid nz: ' + str(nz)
print 'Grid width: ' + str(gridWidth)
print 'Grid height: ' + str(gridHeight)
print 'Grid depth: ' + str(gridDepth)
print 'Grid llc: (' + str(llcx) + ',' + str(llcy) + ',' + str(llcz) + ')

frameNum = int(CmdLineFind("-frameNum", 5))
paddedframe = str(frameNum)
if frameNum < 1000:
paddedframe = "0" + paddedframe
if frameNum < 100:
paddedframe = "0" + paddedframe
if frameNum < 10:
paddedframe = "0" + paddedframe

PI = 3.14159265

llc = Vector(llcx, llcy, llcz)
llci = Vector(llcxi,llcyi,llczi)

print 'Allocating grids'
smokegrid = sparseScalarGrid(nx,ny,nz,gridWidth,gridHeight,gridDepth,llc)
firegrid = sparseScalarGrid(nx,ny,nz,gridWidth,gridHeight,gridDepth,llc)
firecgrid = sparseColorGrid(nx,ny,nz,gridWidth,gridHeight,gridDepth,llc)

print 'Reading data files'

#reading grids
if len(readSmoke)>0:
    print 'reading ' + readSmoke+'.'+paddedframe
    ReadScalarGrid(smokegrid,readSmoke+'.'+paddedframe)
if len(readFire)>0:
    print 'reading ' + readFire+'.'+paddedframe
    ReadScalarGrid(firegrid,readFire+'.'+paddedframe)
if len(readFireColor)>0:
    print 'reading ' + readFireColor+'.'+paddedframe
    ReadColorGrid(firecgrid, readFireColor+'.'+paddedframe)

#Turn grids into fields
smokevolume = gridField(smokegrid)
firevolume = gridField(firegrid)
firevolume = floor(firevolume,.975)
firecolor = gridField(firecgrid)

if len(impactgrid)>0:
    impact = sparseScalarGrid(inx,iny,inz,igw,igh,igd,llci)
    ReadScalarGrid(impact, impactgrid+'.'+paddedframe)
    smokevolume = unions(smokevolume,gridField(impact))

smokecolor = multiply(constant(colorptr(.443,.443,.443,1)),smokevolume)

print 'Reading Camera Data'
readCameraDataFile(str(camfile+'.'+paddedframe),cam)
r.d.initializeRays(cam, width, height)

rd.setScatterVolume(smokevolume,smokecolor)
r.d.setEmissiveVolume(firevolume,firecolor)

print 'Creating DSMs'
lightColor = Color(1,1,1,1)
light2Color = Color(.1,.1,.1,.1)
light = pointlight(Vector(-575,132,511), lightColor, Light.NO_DECAY)
light2 = pointlight(Vector(575,-22,511),light2Color, Light.NO_DECAY)

if len(readdsm1) > 0:
    print 'Reading ' + readdsm1 + '.' + paddedframe
    shadowmap = dsmFile(str(readdsm1 + '.' + paddedframe), light)
else:
    shadowmap = dsm(smokevolume, light, scalarGrid(dsmx,dsmy,dsmz,gridWidth,gridHeight,gridDepth,llc), .1)

if len(writedsm1)>0:
    shadowmap.writeDSM(writedsm1 + '.' + paddedframe)

if len(readdsm2) >0:
    print 'Reading ' +readdsm2 + '.' + paddedframe
    shadowmap2 = dsmFile(str(readdsm2 + '.' + paddedframe), light2)
else:
    shadowmap2= dsm(smokevolume, light2,scalarGrid(dsmx,dsmy,dsmz,gridWidth,gridHeight,gridDepth,llc), .1)

if len(writedsm1)>0:
    shadowmap.writeDSM(writedsm1 + '.' + paddedframe)

rd.addDsm(shadowmap)
r.d.addDsm(shadowmap2)
r.d.addBBox(llc,llc + Vector(gridWidth,gridHeight,gridDepth))

render(rd)

image = Image()
image.reset(width, height, 4)
image.createImage(rd.pixelColors)

imagename = name + "." + paddedframe + "." + extention
writeOIIIOImage(imagename,image,1.0,1.0)

image.reset(width,height,4)
image.createAlphaImage(rd.pixelColors)

imagename = name + 'Alpha' + "." + paddedframe + "." + extention
writeOIIIOImage(imagename,image,1.0,1.0)
Appendix B  Toolkit Scalar Field Node Definitions

Many of these node definitions were found in Physically Based Effects Implicit Function Cheat Sheet. [2]

cone - Creates an implicit cone.

\[
f(x) = \begin{cases} 
0 & x = x_0 \\
height - (x - x_0) \cdot \hat{n} & (x - x_0) \cdot \hat{n} > height \\
(x - x_0) \cdot \hat{n} & (x - x_0) \cdot \hat{n} < 0 \\
\theta - \cos^{-1}\left(\frac{(x-x_0) \cdot \hat{n}}{|x-x_0|}\right) & \text{otherwise}
\end{cases}
\] (1)

cube - Creates an implicit cube.

\[
f(x) = radius - x^p - y^p - z^p
\] (2)

plane - Creates an implicit plane.

\[
f(x) = x \cdot \hat{n}
\] (3)

pyroclast - Creates an implicit pyroclastic sphere.

\[
f(x) = radius - |x| + \text{amplitude} |PN(x)|^{\gamma}
\] (4)

sphere - Creates an implicit sphere.

\[
f(x) = radius - |x|
\] (5)
torus - Creates an implicit torus.

\[ f(x) = 4R_{major}^2 |x \perp|^2 - (|x|^2 + R_{major}^2 - R_{minor}^2)^2 \] (6)

where

\[ x \perp = x - (x \cdot \hat{n})\hat{n} \] (7)

advect - Advects a scalar field by a velocity field and a timestep.

\[ f_{advect}(x) = f(x - U(x)\Delta t) \] (8)

clamp - Clamps the value of a scalar field to a minimum and maximum value.

\[ f_{clamp}(x) = \begin{cases} minVal & f(x) < minVal \\ maxVal & f(x) > maxVal \\ f(x) & \text{otherwise} \end{cases} \] (9)

constant - Creates a scalar field with a constant value

\[ f(x) = c \] (10)

cutout - Removes the second scalar field from the first

\[ f_{cutout}(x) = \min(f(x), -g(x)) \] (11)
floor - Sets areas of a scalar field that are less than a minimum value to 0.

\[
f_{\text{floor}}(\mathbf{x}) = \begin{cases} 
0 & f(\mathbf{x}) < \text{minVal} \\
 f(\mathbf{x}) & \text{otherwise}
\end{cases}
\]  

(12)

intersection - Creates a scalar field that is positive only where both of the input scalar fields are positive.

\[
f_{\text{intersection}}(\mathbf{x}) = \min(f(\mathbf{x}), g(\mathbf{x}))
\]  

(13)

mask - Returns 1 anywhere the scalar field is positive and 0 everywhere else.

\[
f_{\text{mask}}(\mathbf{x}) = \begin{cases} 
1 & f(\mathbf{x}) > 0 \\
0 & \text{otherwise}
\end{cases}
\]  

(14)

rotate - Rotates a scalar field by \(\theta\) degrees around an axis.

\[
f_{\text{rotate}}(\mathbf{x}) = f(\mathbf{x} \cos \theta + (\text{axis}(\mathbf{x} \cdot \text{axis}))(1 - \cos \theta) + (\mathbf{x} \times \text{axis}) \sin \theta)
\]  

(15)

scale - Uniformly scales a scalar field at the origin.

\[
f_{\text{scale}}(\mathbf{x}) = f(\mathbf{x}/s)
\]  

(16)

translate - Translates a scalar field.

\[
f_{\text{translate}}(\mathbf{x}) = f(\mathbf{x} - \mathbf{v})
\]  

(17)
union - Combines two scalar fields into one.

\[ f_{\text{union}}(x) = \max(f(x), g(x)) \]  

(18)

warp - Displace a scalar field by a vector field.

\[ f_{\text{warp}}(x) = f(x + U(x)) \]  

(19)
Bibliography


