Clemson University
TigerPrints

All Theses | Theses

5-2014

MeshPotato: A C++/Python API for Production Volumetric Rendering

Kevin Coley
Clemson University, kcoley@g.clemson.edu

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

Part of the Computer Sciences Commons, and the Fine Arts Commons

Recommended Citation

Coley, Kevin, "MeshPotato: A C++/Python API for Production Volumetric Rendering" (2014). *All Theses*. 1940.
https://tigerprints.clemson.edu/all_theses/1940

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.

MeshPotato: A C++/Python API for Production Volumetric Rendering

---

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
Kevin Arnold Coley Jr
May 2014

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

Abstract

MeshPotato is a production volume rendering API written in C++. Its purpose is to simplify the creation of high quality volumetric effects such as fire, smoke, clouds and explosions. MeshPotato has been designed to be extensible and flexible for quick changes. Python bindings have been implemented with this library to allow for tools that are scripted and integrated within popular 3D modeling applications such as Maya and Houdini. The design of MeshPotato is discussed along with its plugin system, volume rendering API and some results from using the tool.

Acknowledgments

This thesis would not be possible without the background knowledge and guidance from Dr. Jerry Tessendorf and his Production Volume Rendering course. I also would like to thank him for serving as the chair of my committee. I would also like to thank Zhaoxin Ye and Karen Stritzinger for providing sample images, proofreading this thesis and testing MeshPotato. Tim Curtis contributed his MeshViewer tool to the MeshPotato project. Sam Casacio provided input on OpenVDB support. Ken Museth gave me input on designing and accelerating the volume rendering process. Jon Barry came up with the name for the tool. Yujie Shu introduced the concept of deep images to me. Kaitlyn Li motivated me in the writing of my thesis. I appreciate all the feedback from the DPA 891 group in providing design considerations for MeshPotato. I thank Dr. House and Dr. Davis for not only serving as members on my committee but also giving me the background knowledge I would need to implement my thesis. I appreciate the confidence from the Digital Production Arts faculty and students which motivated me to challenge myself. Finally, I would like to thank my family for believing in me and supporting me financially.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Acknowledgments . . . . **iii**

List of Tables . . . . **v**

List of Figures . . . . **vi**

List of Listings . . . . **vii**

1 Introduction **1**

2 Background **3**

3 The MeshPotato Architecture **14**

4 Volume Rendering **19**

5 Python Bindings **24**

6 Results **27**

7 Conclusions and Discussion **31**

Appendix **32**

Bibliography **36**

List of Tables

3.1 Classification of data types in MeshPotato 15

4.1 Classification of volume types in MeshPotato 20

List of Figures

1.1 Volumetric fire from the motion picture *The Hobbit: An Unexpected Journey* released in 2012 from Weta Digital's FX framework, Synapse. 1

2.1 The ray marching process. A ray is emitted from the camera through an image plane pixel. Samples are taken within the volume to compute the total accumulated light information
for the particular pixel 4
2.2 CSG operations on two spheres 5
2.3 Types of Rectangular Grids, from *Volumetric Methods in Visual Effects* [14] 6
2.4 Light leaking along right edge of frustum. Image from *Volumetric Methods in Visual Effects* [14] 7
2.5 Types of Frustum Grids, from *Volumetric Methods in Visual Effects* [14] 8
2.6 Narrow band representation for a sphere in OpenVDB 8
2.7 Fog Volume representation for a sphere in OpenVDB 9
2.8 RGB lighting for the VDB cloud bunny 9
2.9 Deep Image of Cloud Bunny 10
2.10 Static libraries figure from *API Design for C++* [9] 11
2.11 Dynamic libraries figure from *API Design for C++* [9] 12
2.12 Plugin library figure from *API Design for C++* [9] 13

3.1 Overview of the MeshPotato Architecture. The blue represents core classes the green represents modules, and the red represents tools built from MeshPotato . . 14
3.2 Loading a plugin into the Plugin Factory 17
3.3 Creating a Plugin 18

4.1 The Volume Type Hierarchy 19
4.2 Union of Pyroclast and Sphere 21
4.3 Pyroclasts with different frequency parameters 23

5.1 Rendered result from the pyroLoop function 26

6.1 MPConvert for Maya, MeshViewer and vdb_view 28
6.2 Zhaoxin's Cloud Modeling workflow using MPConvert for Maya 29
6.3 The OpenVDB public cloud bunny, rendered using MPVolumeRender. 30

Listings

4.1 Union of Two Volumes using Resolution Independent Volumes 21
5.1 Exposing the Color class to Boost.Python 24
5.2 Creating a Sphere in Python 25
5.3 Function example in Boost.Python wrapper interface 25
5.4 Using reference counting variables to create complex volumes 26
1 VDBOutputPlugin.h (part 1) 33
2 VDBOutputPlugin.h (part 2) 34
3 VDBOutputPlugin.h (part 3) 35

Chapter 1

Introduction

Figure 1.1: Volumetric fire from the motion picture *The Hobbit: An Unexpected Journey* released in 2012 from Weta Digital's FX framework, Synapse.

Production volumetric rendering is the technique of modeling and rendering volumetric density and color in order to create images for film or animation. Some examples of volume rendering include smoke, fire, clouds, tornadoes and splashes (Figure 1.1). Production volume rendering is a specific application of the broad field of volume rendering, which also includes medical imagery

and scientific visualization. Little information is published on production volume rendering concepts since most of the research occurs in visual effects studios. To address this issue, several individuals from various production studios presented a SIGGRAPH course in 2010 on this field [14].

Digital Domain, Dreamworks Animation, Double Negative, Sony Imageworks, Rhythm & Hues, and Weta Digital are some of the studios with proprietary in-house volume rendering tools [14]. These tools were written for artists to allow them to create sophisticated effects. Until recently, production volume rendering software was only publically available in commercial packages such as Autodesk Maya, Sidefx Houdini and Pixar's Photorealistic RenderMan. In 2009, Sony Imageworks released an open source volume file format called Field3d [12]. Magnus Wrenninge later released one of the first open source production volume renderers on github called *PVR*[1]. Also in 2012, Dreamworks Animation publically released their production volume rendering C++ library called OpenVDB which consists of a hierarchical sparse grid data structure and a suite of tools for volume rendering [7].

Though these open source products provide useful sets of tools for creating volumetric effects, a higher layer of abstraction must be applied to allow artists to focus on enriching the visual quality of the images, get a result quickly for further iterations, and hide the underlying implementation details. With this in mind, and with the availablility of open source software, it is now possible to author a production volume rendering API which is free of license restrictions, is fast and simple to use and modify. This thesis presents the design and implementation of the volume modeling and rendering production tool, MeshPotato. Background information in Chapter 2 defines some of the concepts in this paper. The design of MeshPotato is discussed in Chapter 3, and the approach to volume rendering in Chapter 4. The Python binding interface in Chapter 5 leads to the discussion of tools derived from MeshPotato in Chapter 6. The use of MeshPotato by artists in their productions is also demonstrated in Chapter 6.

[1] *PVR* is a production volume renderer written for Wrenninge's book *Production Volume Rendering*. *PVR* uses Field3D as well as Python bindings using Boost.Python and is publically available on http://github.com/pvrbook/pvr [13]

Chapter 2

Background

In the context of volume rendering, a volume is a field which contains information in space. This information could be density, color, light intensity, velocity, and other properties associated with volumes. To display the volume requires an application that can produce an image from the data. A camera is usually placed somewhere in space and is aimed at the volume. From the camera, rays are emitted through a set of pixels within an image plane. The rays will step through the field data, accumulating light information at each step. Eventually, once the ray traverses through the volume, the data will be used to produce a color for the particular pixel (Figure 2.1). This process is called ray marching and it is a discretized single-scatter approximation of the fuller theory of radiative transfer [11]. The equation for accumulating light in a ray march is

$$L(\mathbf{x}_c, \mathbf{n}_p) = \sum_{j=0}^{M-1} T_j \int_0^{\Delta s} ds\ C^T(\mathbf{x}_j + s\mathbf{n}_p)\ \rho(\mathbf{x}_j + s\mathbf{n}_p)\ \exp\left\{-\int_0^{s} ds' \kappa\ \rho(\mathbf{x}_j + s'\mathbf{n}_p)\right\} \tag{2.1}$$

A ray $\mathbf{x}(s)$ is emitted from a camera positioned at $\mathbf{x}_C$ aimed in the direction of a pixel $\mathbf{n}_p$ and measures a distance s to a point in the volume: $\mathbf{x}_j = \mathbf{x}_C + j\Delta s\mathbf{n}_p$. The ray progresses through an interval Δs along the direction $\mathbf{n}_p$. At each point along the ray, the total color, C^T, from the emission and scattering, κ, of the volume is accumulated and multiplied against the density, ρ, of

the volume. The transmissivity factor, T_j, is

$$T_j = \prod_{k=0}^{j-1} \exp\left\{ -\int_0^{\Delta s} ds \ \kappa \ \rho(\mathbf{x}_k + s\mathbf{n}_p) \right\} \tag{2.2}$$

The volume generally would be rendered from a finite region of space, $s_{max} - s_0$, which can be used to set the begin and end points of the ray. The interval is broken into M steps each with a length of Δs such that $M\Delta s = s_{max} - s_0$;

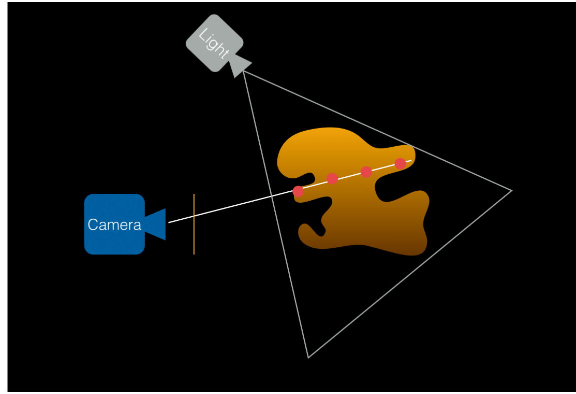
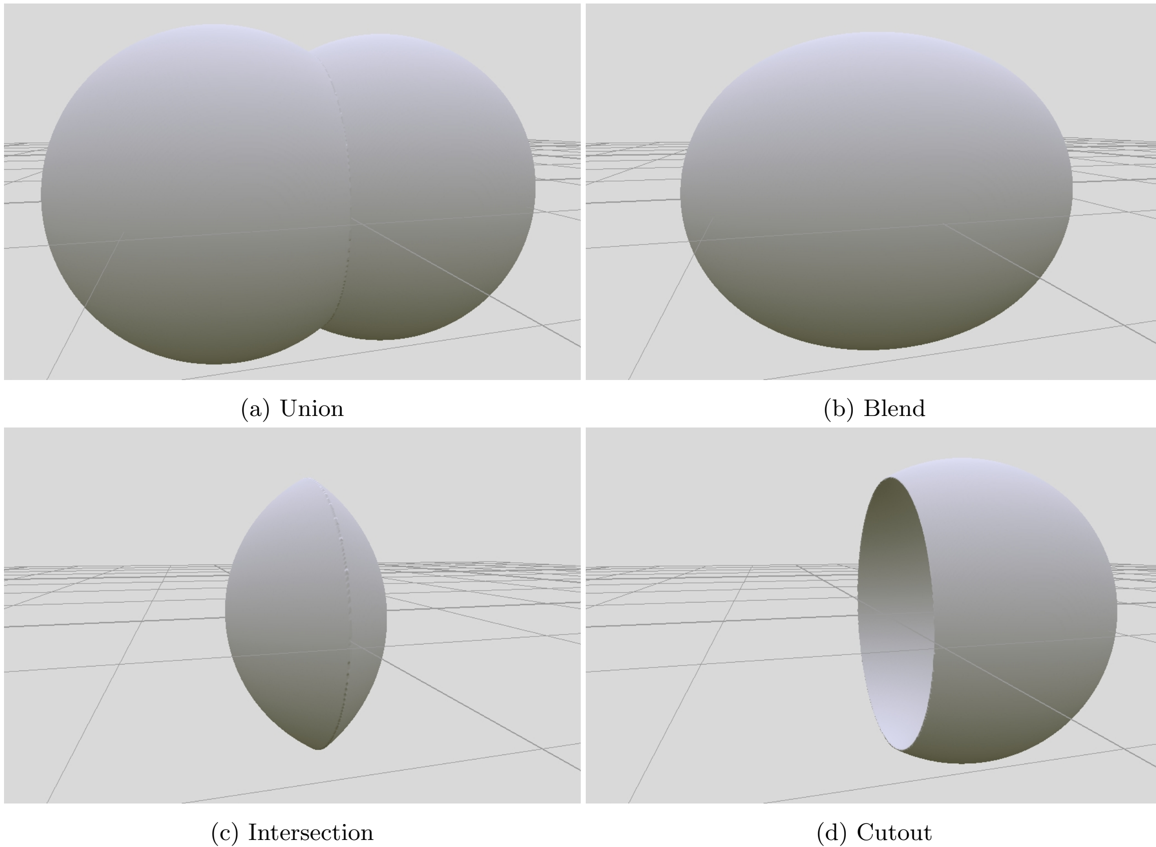


Figure 2.1: The ray marching process. A ray is emitted from the camera through an image plane pixel. Samples are taken within the volume to compute the total accumulated light information for the particular pixel

A volume can be represented in various ways. One form is as an implicit function. An implicit function evaluates to a value of zero on its surface. Another useful property of implicit functions is that constructive solid geometry (CSG) techniques can be applied to them in order to create complex new functions (Figure 2.2). CSG is a technique used to model surfaces through the use of binary operations [14]. A limitation of implicit functions is that a bounds may not easily be defined depending on their complexity. Another disadvantage is that whenever its value in a position in space is needed, it has to be calculated, which can be computationally expensive. One solution is to sample the implicit function onto a 3D grid.



(a) Union

(b) Blend

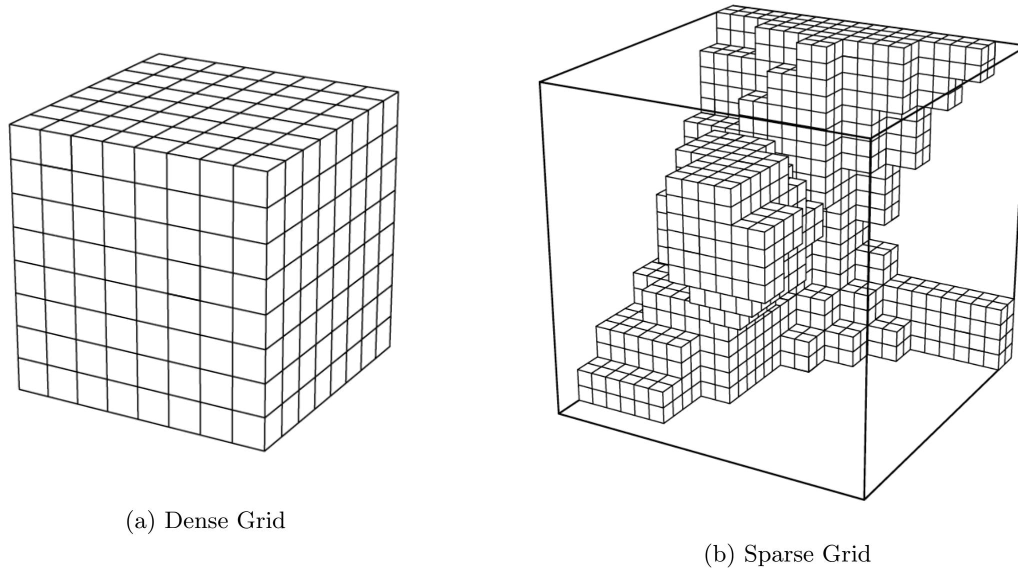
(c) Intersection

(d) Cutout

Figure 2.2: CSG operations on two spheres

3D Grids are data types for storing and accessing data that is represented in three-dimensional space by partitioning space into discrete units called voxels. Some volumetric algorithms require a grid to be implemented. Grids can also accelerate the rendering process since the data has been precomputed and only needs to be accessed. Even though a grid is created with discrete samples, information can be obtained from arbitrary positions through interpolation [10]. Grids are structured in various forms. The simplest structure is a grid where the number of voxels is predefined and are allocated at construction, or dense grid (Figure 2.3a). However, the resolution of this grid would be limited by the amount of memory available on a particular system. Many volumes also do not occupy the full range of space in the grid so most of the memory could be unused. A compromise to the dense grid is defining a structure where the memory is dynamically allocated when needed, or sparse grid (Figure 2.3b). Sparse grids can allow higher resolutions to be defined, assuming the volume does not occupy the entirety of the space it defines. Storing data in a sparse grid can be slower since it must first check to determine whether memory has been preallocated at a certain

position, and reserve memory when needed.



(a) Dense Grid

(b) Sparse Grid

Figure 2.3: Types of Rectangular Grids, from *Volumetric Methods in Visual Effects* [14]

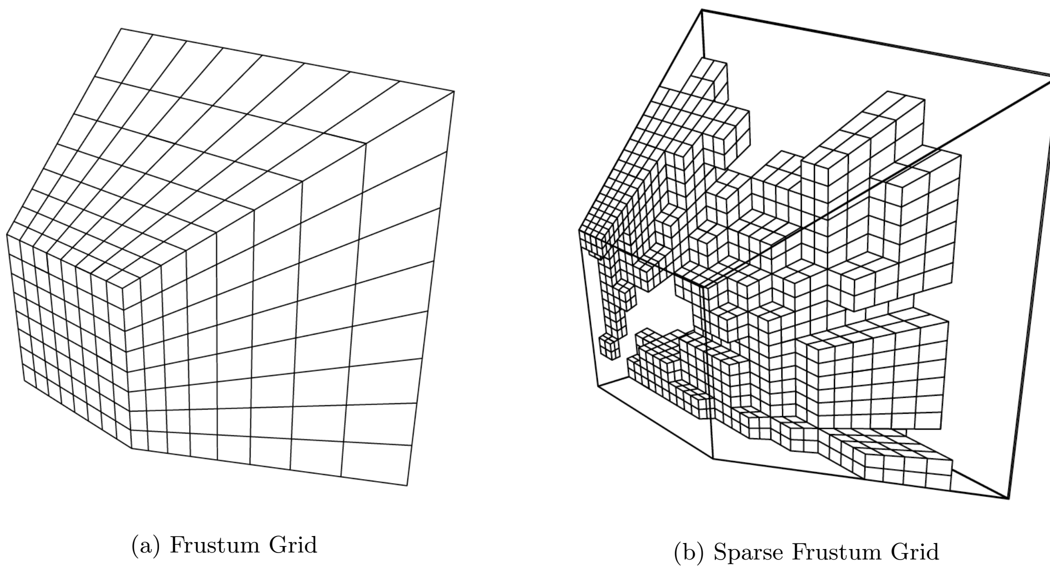
A frustum grid is a grid mapped to the shape of a camera frustum. The idea is that data outside the field of view of a camera would not necessarily be useful. Frustum grids are convenient for computing deep shadow maps (DSMs) for a light shaped like a camera frustum (Figure 2.5). DSMs is a technique for quickly computing shadows, by taking into account the fractional visibility of light at different levels of depth [6]. In computing frustum DSMs, the resolution of the grid in some circumstances does not need to be very high, which accelerates computation time. The drawback is that voxels will grow in size they further they are from the near plane of the camera and can cause issues such as light leaking since volume density is not available outside the frustum shape (Figure 2.4). Finally, a grid can be written in such a way that the resolution can change over time and would not need to be predetermined. This can be useful for running time-varying simulations and is a feature of OpenVDB.

OpenVDB is a hierarchical data structure which can represent sparse data with dynamic topology within a 3D grid and also a set of tools for manipulating the data. The grid structure is able to represent high-resolution sparse volumes and provides fast constant time random access for insertions, deletions and removals from the grid structure [7]. At its lowest level it is a tree which contains tiles and leaf nodes. As values are added to the grid, its overall structure changes

Figure 2.4: Light leaking along right edge of frustum. Image from *Volumetric Methods in Visual Effects* [14]

dynamically as opposed to an axis-aligned grid with a predefined resolution. OpenVDB is flexible enough to fulfill all of the properties of the previously mentioned grids, allowing for the quick generation of high resolution volume data in various forms.

OpenVDB uses the concept of a narrow band to represent a levelset. A levelset can represent a volumetric surface as a function in which its evaluation results in a signed distance. This signed distance indicates whether a spatial point is inside or outside of the surface [14]. A narrow band is a shell-like region around the boundary of a volume for which levelset values are stored. The voxels to the exterior of the narrow band are inactive and evaluate to a constant positive number representing the background value. The values enclosed by the narrow band are also inactive and evaluate to the negative of the background value. The sign indicates whether the voxel lies above or below the surface of the volume (Figure 2.6).



(a) Frustum Grid

(b) Sparse Frustum Grid

Figure 2.5: Types of Frustum Grids, from *Volumetric Methods in Visual Effects* [14]

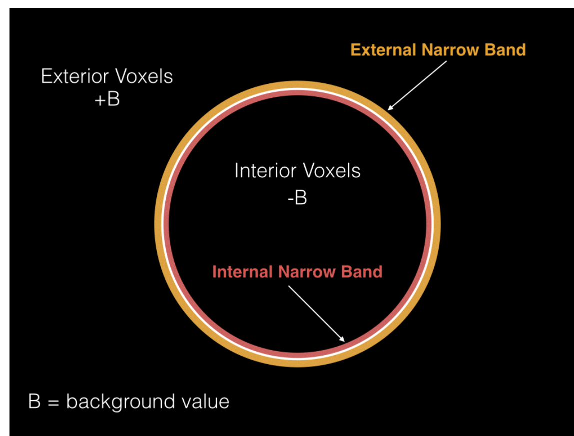


Figure 2.6: Narrow band representation for a sphere in OpenVDB

Since the voxel data is active only inside the narrow band, it will have to be modified in order to be rendered as a volume. OpenVDB provides a function to convert the narrow band into a fog volume by setting the internal voxels to a constant value of 1.0, the external voxels to 0.0 and by applying a gradient to the narrow band voxels to produce a smooth transition from the internal to the external voxels (Figure 2.7).

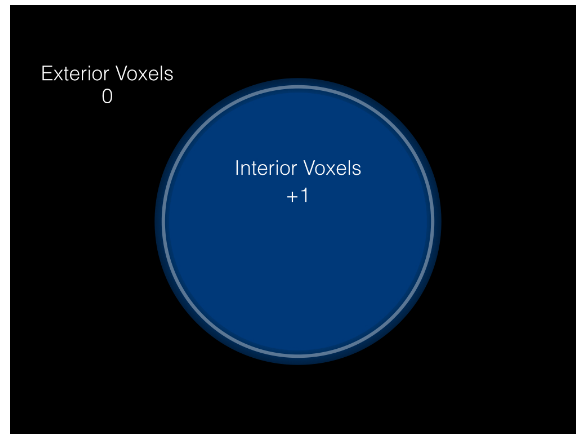


Figure 2.7: Fog Volume representation for a sphere in OpenVDB

The resulting image from a volume renderer can contain useful information that can be modified during compositing in order to avoid re-rendering. One production technique is the use of RGB lighting, where the key, fill and rim lights are each assigned to a specific color channel in the final image [5]. The color channels can then be separated, color corrected and merged back together again to produce the final image (Figure 2.8). The caveat is that the volume color must be spatially constant for this to work properly.

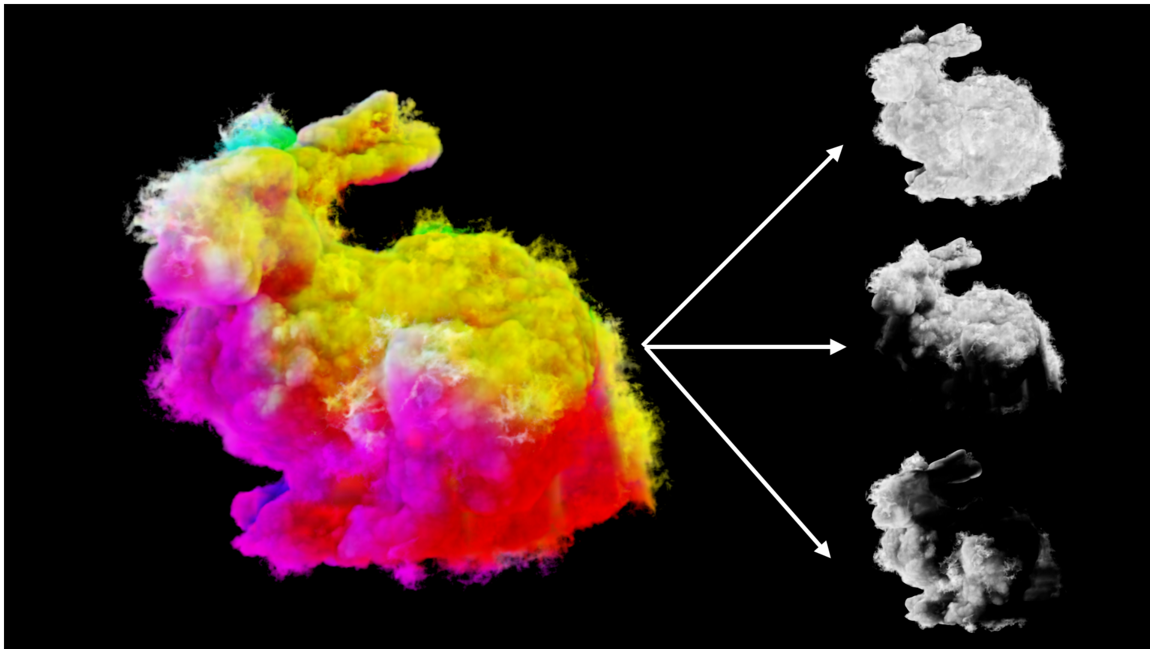
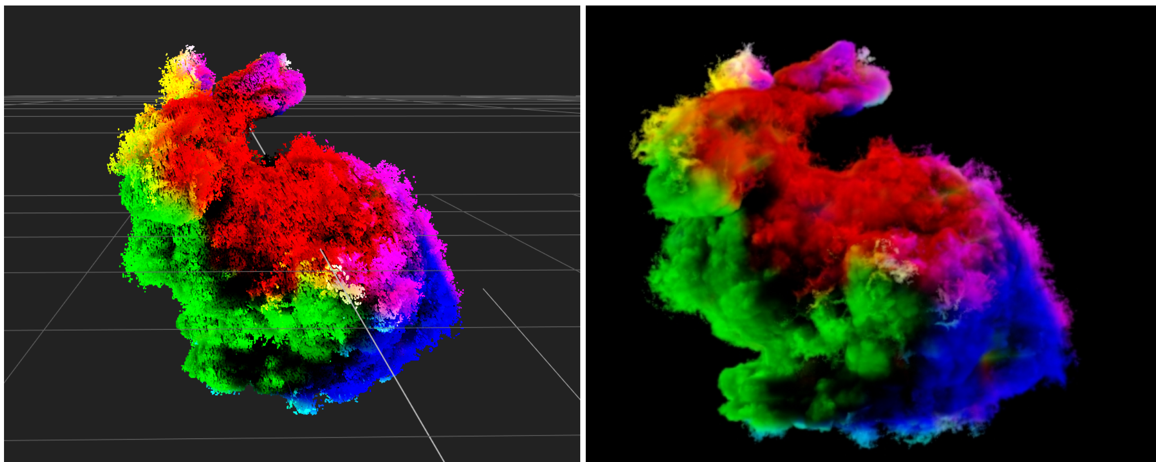


Figure 2.8: RGB lighting for the VDB cloud bunny

Another technique is the use of a deep image format to store multiple color and depth values per pixel. This format can generate hold out mattes on the fly, allow for translations in depth along the camera z-axis, and can be used for depth of field effects. Currently, OpenEXR 2.0 provides an open source deep image format which supports these types of operations. Nuke, a compositing package by The Foundry, can interpret this format and apply the aforementioned compositing operations (Figure 2.9). Deep images tend to be larger than a 2D image, depending on the amount of samples stored in the image. However, when used efficiently, they reduce the need for rerendering or generating holdout mattes.



(a) Deep Image Points of Bunny (b) Deep Image of Bunny

Figure 2.9: Deep Image of Cloud Bunny

To generate a deep image, samples in the raymarching process can be stored into pixel buffers and then later stored within the deep image file. Each sample contains the total color at the point and a depth from the rendering camera. The color is the product of the material color and the color from the light. The alpha channel for the color is computed by subtracting the transmissivity from one. The depth is the distance from the rendering camera. Two deep images can be merged by simplying concatenating overlapping samples together. A deep image can be "flattened" or converted into a 2D image by applying an over composite to each sample, starting from the point closest to the camera, assuming the samples are sorted based on their depth. Deep holdouts generate a "hole" in the volumetric data, removing information for where the deep samples from the second image are located. This technique has been documented at Weta Digital [3] as well as Industrial Light and Magic [4].

It is often useful to wrap a C++ tool with Python. Python is a scripting language that is widely used in the industry [9]. It can be more convenient to use Python due to its portability and flexibility, as opposed to a compiled language like C++. Python bindings can be generated for a C++ tool using a wrapper library. A Python function, for example, can be made up of several C++ functions to simplify Python scripting. Function and class names can also be changed to conform to Python coding standards such as PEP-8.

Two popular wrapper libraries are SWIG and Boost.Python. SWIG is a code generator, which automatically generates code from C++ header files [2]. The wrapping interface uses a set of rules to generate the mappings into Python. Due to the automatic code generation, SWIG bindings can be difficult to debug. Boost.Python is an interface library which requires every mapping between C++ and Python to be explicitly declared [1]. While this can be potentially time consuming for large libraries, the advantage is that the mapping is simpler to debug since it is essentially compiling C++ code.

When writing C++ libraries, the source code is often stored in static or dynamic libraries. Static libraries are collections of object files that are used when linking to an application. All of the library code is archived within a single file. Any user of a static library must embed a copy of the code into all of the applications which use it. Since the code is within the application, code can be distributed without having to worry about locating dependencies during run-time. However, if the library is large, the applications could potentially occupy excessive amounts of memory. Also, the entire application would have to be recompiled if the library needs to be updated (Figure 2.10).

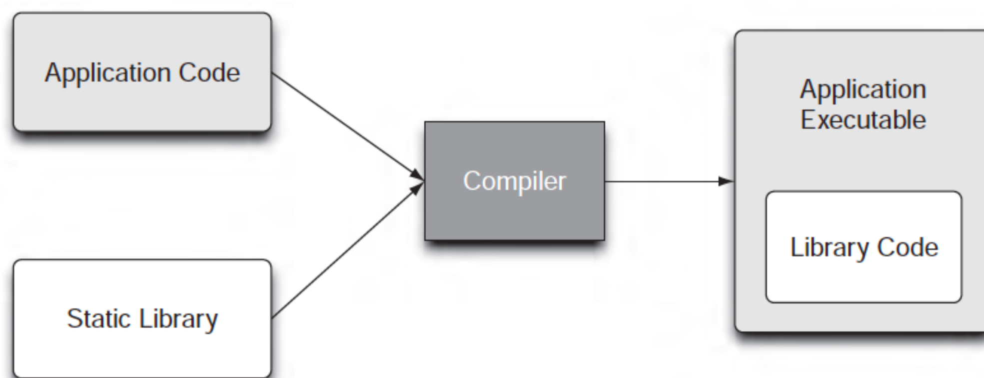


Figure 2.10: Static libraries figure from *API Design for C++* [9]

Dynamic libraries are linked against during the runtime of the application (Figure 2.11). Dynamic libraries must be distributed with the application so the program can execute during runtime. If it links against other dynamic libraries, they must also be available on the system. One advantage of dynamic libraries is that they can occupy less disk space than static libraries since multiple applications can link to it during runtime. Another benefit is that they can be replaced with newer, backwards compatible versions without having to recompile the main application [9].

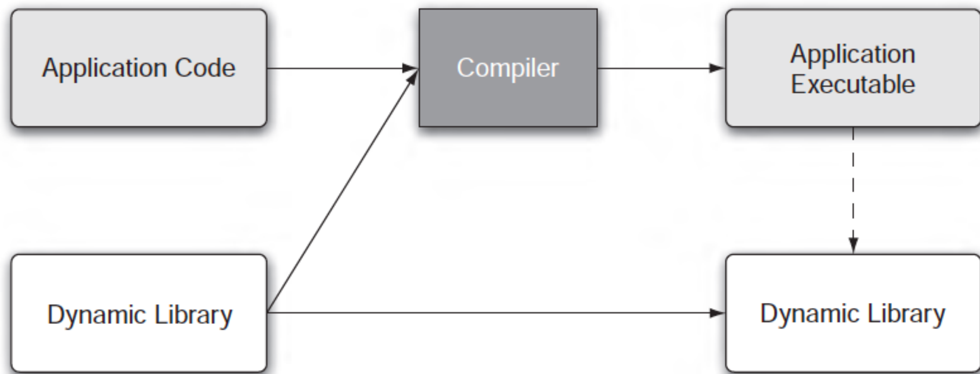


Figure 2.11: Dynamic libraries figure from *API Design for C++* [9]

Shared libraries can also be used as plugins and be loaded by the main application on demand (Figure 2.12). This allows developers of an application the flexibility of writing custom plugins and placing them in a location in which the application can locate them [9].

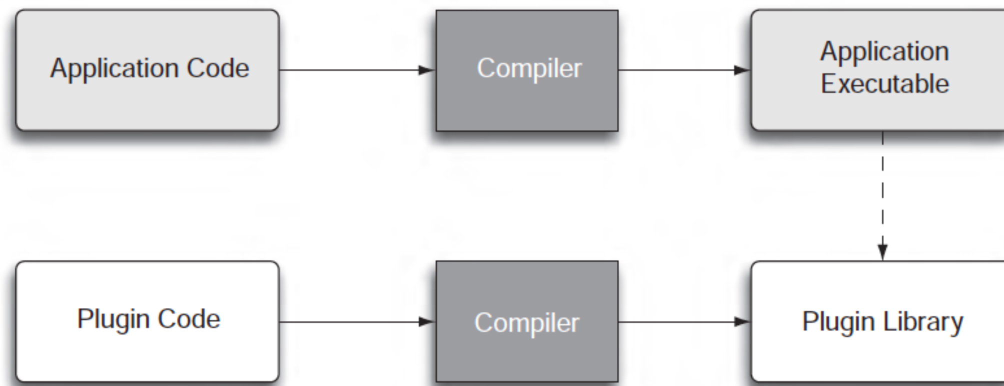


Figure 2.12: Plugin library figure from *API Design for C++* [9]

Chapter 3

The MeshPotato Architecture

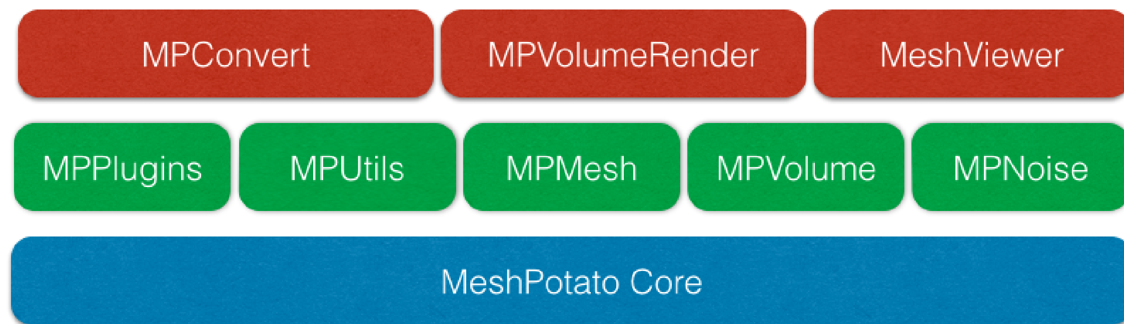


Figure 3.1: Overview of the MeshPotato Architecture. The blue represents core classes the green represents modules, and the red represents tools built from MeshPotato

In the development of MeshPotato, it was important for the tool to be usable in a production environment where unpredictable changes need to be made to accomplish a visual effect. For this to be possible, MeshPotato had to have a stable set of classes and utilities but also be flexible to modification. The structure of MeshPotato needed to be simple to understand to allow for future changes to the framework, and offer the ability to allow modification of the existing structure. Figure 3.1 gives an overview of the entire architecture.

A design decision was to classify portions of MeshPotato and identify them through the use of namespaces. This not only prevents name clashes with other third party libraries, but also identifies the context of a particular object. For example, a camera is considered a utility within

MeshPotato. Thus it exists under the MPUtils namespace. The full class name would then be MeshPotato::MPUtils::Camera. The Volume<T> class is within the MPVolume namespace. It will be scoped as MeshPotato::MPVolume::Volume<T> . MPPlugins contains all the classes associated with the plugin system. MPUtils is a collection of useful routines such as vectors, cameras, matrices and attribute tables. MPMesh deals with reading and writing mesh data. MPVolume encapsulates the volume classes, grids and the volume rendering classes. MPNoise is a set of utilities for generating procedural noise. Table 3.1 shows all the classifications of MeshPotato.

Classification	**Description**
MPPlugins	Manages the Core Plugin System
MPUtils	Utility classes such as Vectors, Cameras, etc
MPMesh	Classes for working with Mesh Data
MPVolume	Classes for working with Volume Data
MPNoise	Classes for working with Noise

Table 3.1: Classification of data types in MeshPotato

With scoping in place, the next issue to address is the ability to extend MeshPotato without having to change the code base. An approach used in many production packages is to use a plugin system, which allows users to add custom functionality that does not currently exist within the application. In MeshPotato, the plugin system consists of a plugin manager for loading and unloading plugins, a collection of factory classes for registering the plugins, and a plugin API for developers to be able to write them. The types available for plugins are written as abstract virtual base classes, which provides a blueprint for what methods need to be implemented in order to create a plugin. The supported types are dependent on the types available in the plugin factory classes. Enforcing certain base classes to the plugin system insures type safety and simplifies inheritance within C++ and Python. This approach is similar to OpenImageIO, a library for reading and writing images, where plugins are loaded automatically based on an image file name. An example plugin is listed in Appendix A.

In the case in which MeshPotato would need to be updated and extensions through plugins are not sufficient, it becomes important to have a versioning system so that developers of derived tools would be aware of any changes that would require them to be updated. MeshPotato follows the convention of MAJOR.MINOR.PATCH system. A change in the MAJOR number indicates new functionality in the API that may require changes to derived code. A change in the MINOR number

introduces new functionality but should not require a change in any code using MeshPotato. Finally, the PATCH update indicates a bug fix and also should not require any changes.

Once the core functionality has been put in place, the next consideration is how to structure the code so that it is readable and only exposes the necessary information. MeshPotato employs the Pimpl idiom, which hides the implementation details within a private member of the class. The advantage of this approach is that the header files only contain the accessible public methods. Another advantage is that the private implementation variables would only need to be modified within the implementation files as opposed to the having to change the header files [9].

The plugin system is defined by a plugin manager, which is in charge of the registration and loading of plugins. It keeps track of registered plugins through the use of factory classes. When the plugin class is used, a request is made to the factory to load the plugin. The factory then uses the plugin manager to locate the plugin so it can register it.

One issue that arises in this situation is the management of attributes from classes from plugins. MeshPotato is not able to predict what parameters are required for artist designed plugins. To work around this limitation, MeshPotato provides an AttributeTable class, which stores arbitrary variables. These variables can then be optionally passed in to the plugin class as an argument, overriding the default values defined by the plugin. If the plugin were to update and support more variable types, the user would simply add that parameter to a custom attribute table if he or she wished to override it.

When using the plugin system, MeshPotato must first search for plugins during runtime. It uses an environment variable, MESHPOTATO_PLUGIN_PATH as the search directory for plugins. Any shared library within the directory will be checked to see if it can be registered. After registering the plugins, the artist would them load them into the application in order to use them (Figure 3.2). Since the plugins are located during runtime, it is possible for a new plugin to be written and placed in the search directory before the application runs. This is useful for a studio environment where a new data type can be shared across a collection of tools.

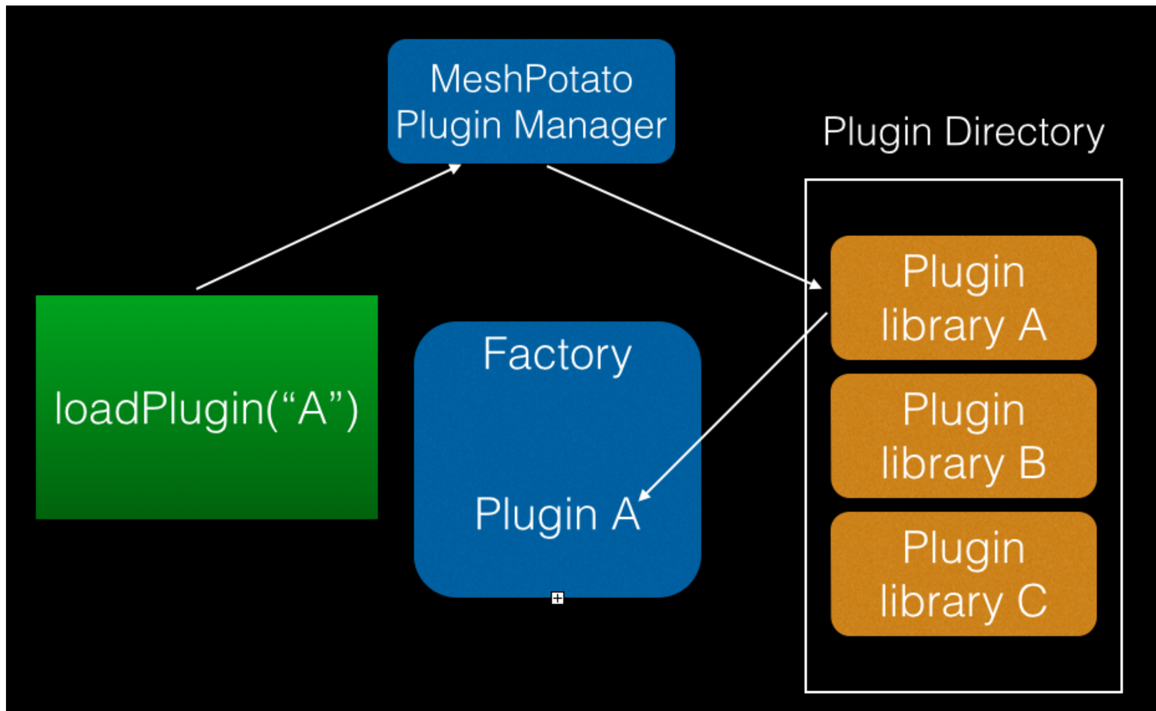


Figure 3.2: Loading a plugin into the Plugin Factory

Writing a plugin to work with MeshPotato is relatively simple. Each plugin requires a set of four plugin functions to be implemented. One function creates a new instance of the plugin object, which should be derived from a base class in MeshPotato. The second routine is responsible for deleting the instance. The third routine registers a display name. Finally, the init routine registers the plugin with MeshPotato. The artist would then implement the virtual functions in the derived plugin. Finally, the plugin would have to be compiled against MeshPotato (Figure 3.3). For the plugin to be available, it would have to be located in the search path of MESHPOTATO_PLUGIN_PATH.

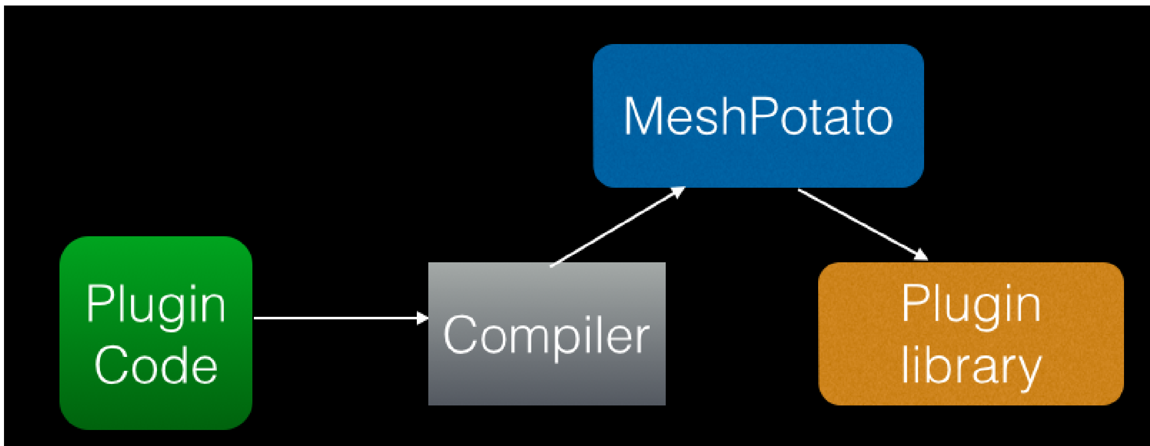


Figure 3.3: Creating a Plugin

Several plugins have already been implemented since they are used in MPConvert, a mesh conversion tool, and MeshViewer, a mesh visualization tool. These tools will be discussed in Chapter 7.

Chapter 4

Volume Rendering

The volume rendering API is designed to provide an interface for modeling and rendering volumetric effects. An artist working with this interface would typically want to be able to work with many parameters to have control over the quality of the geometry or image. Also, it is important to have an efficient tool where the result can be viewed relatively quickly.

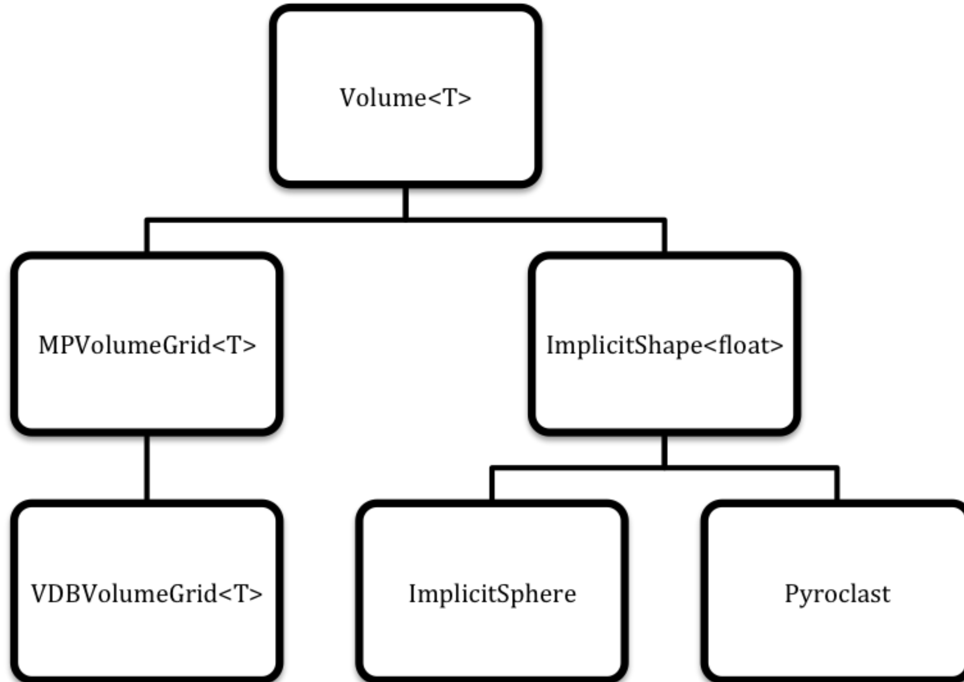


Figure 4.1: The Volume Type Hierarchy

Volume<T> is a templated abstract base class for almost all of the classes within the scope of MPVolume (Figure 4.1). The Volume<T> class defines two pure virtual functions, eval and grad. eval returns an interpolated value at a particular position in space, and grad returns the gradient of a value in space. The return type of these functions depend on the type of volume it represents. If the volume represents a set of floating point numbers, such as Volume<float>, its return type for eval is identical to the type of the volume. The return type for the grad function, which represents the gradient, would be a vector. If the volume defined a vector field such as Volume<MPUtils::MPVec3>, the return type of eval would also be a vector and the grad function would return a matrix. All other types default to returning an integer as default and can be ignored. For example, if the volume represented a color field such as Volume<Color>, it would not make sense to return a gradient solely on color information. Other classes can then be derived from Volume<T>. The supported volume types and the expected return values are listed in (Table 4.1).

volume type	**eval**	**grad**
float	float	MPVec3
MPVec3	MPVec3	MPMat3
MPMat3	MPMat3	—
Color	Color	—

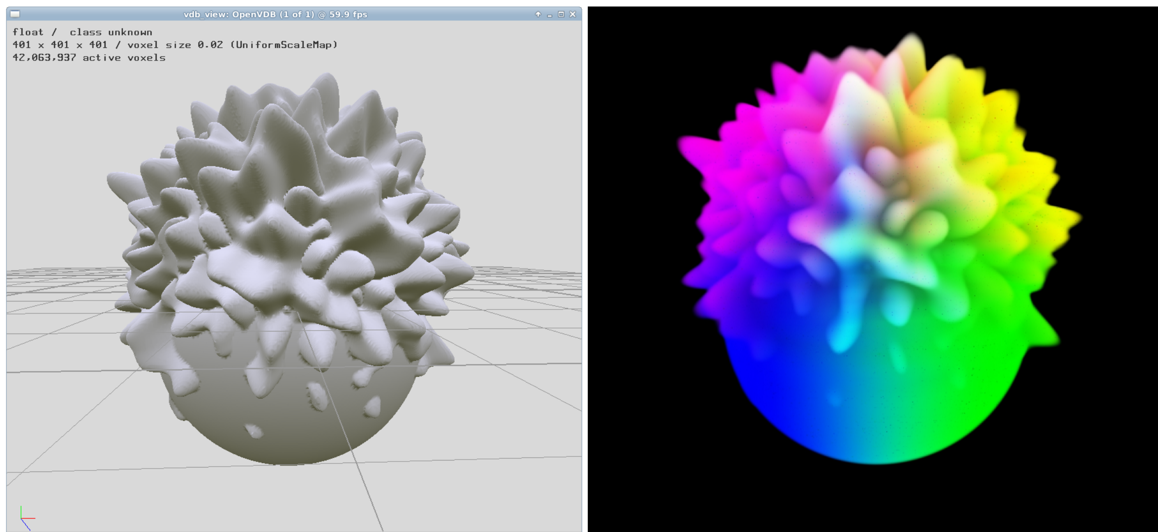
Table 4.1: Classification of volume types in MeshPotato

The reason behind this approach is that the volume classes make use of resolution independent volumes [11]. Resolution independent volumes have two main properties. For one, it is possible to mix between gridded and non-gridded volume types since the complexity of the implementation is implicitly defined within the data structures. This can allow for operations such as addition to be applied to two arbitrary volumes of the same type. For example, the addition of two volumes in MeshPotato of the same type can be executed through the class AddVolume<T>. Fields with different properties can also be combined, regardless of the internal resolution or size. This is desirable for several reasons. One crucial reason is that the code is cleaner to work with when implementing algorithms since the format details are handled automatically. It is efficient since calculations only need to be computed where and when needed. High resolution procedural algorithms can be applied to volumes to produce finite detail. Many algorithms can be implemented without the need for a

grid, with the trade-off that the computational time can be quite lengthy since each position in space would require iterating through a sequence of calculations, but this can be eliminated by caching the information into a grid strategically .

Listing 4.1: Union of Two Volumes using Resolution Independent Volumes

```
1 MPUtils::Noise_t noise = MPUtils::Noise_t();
2 noise.frequency = 5.0
3 noise.amplitude = 4.0
4 VolumeFloatPtr sphere1 = PyroclasticSphere(2.0, MPVec3(0.0, 0.5, 0.0), noise);
5 VolumeFloatPtr sphere2 = ImplicitSphere2(1.0, MPVec3(0.0, -0.5, 0.0));
6 VolumeFloatPtr unionSpheres = UnionVolumeFloat(sphere, sphere2);
```



(a) OpenVDB levelset in vdb_view (b) Rendered volume from MeshPotato

Figure 4.2: Union of Pyroclast and Sphere

With resolution independent volumes, MeshPotato performs high level modeling operations on implicit functions like CSG operations. For instance, a pyroclastic sphere can be unioned with a normal sphere. This gives artist a simple syntax to work with in order to create complex volume shapes. Listing 4.1 is an example of the union of two different volumes and Figure 4.2 is the result.

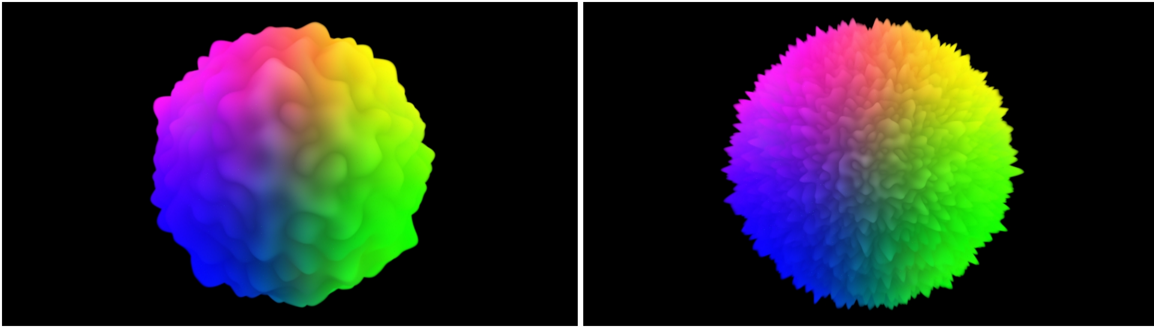
The volume rendering API takes advantage of OpenVDB for its grid type, with the option of registering new grid formats by deriving from Volume<T>. After the creation of resolution independent volumes, the data can be stamped into a VDB grid based on a specified bounding region and voxel size. The VDB grid is especially useful during the raymarching process since it is able to detect active leaf nodes and only march through regions where values of interest exist. Lighting is accomplished through the use of DSMs. MeshPotato uses the frustum transform in

OpenVDB along with a camera to create a frustum grid structure. MeshPotato has functionality to automatically compute the view, near and far planes for a frustum camera aimed at a VDB volume by aiming at the center of its bounding box and determining the intersections where the near and far plane would encapsulate the volume. Alternatively, an artist can manually place the lights anywhere in space. MeshPotato also has a raymarcher which takes advantage of the Volume Ray Intersector in OpenVDB. The Volume Ray Intersector has a march routine which returns intervals along a ray where active values are present. An artist can then specify a step size and the intersector will march through these intervals to compute the color information. Afterwards, the march routine can continuously be called until no active values are left along the path of the ray. This greatly accelerates the raymarching process since it takes advantage of the sparse structure of the VDB grid. Also, the VDB raymarcher is multithreaded, with the option to limit the maximum number of threads using TBB [1].

MeshPotato also has a generic raymarcher that can handle gridless and gridded volumes. Though slower, it can be useful for rendering fine detail from techniques like gridless advection. Additionally, it can still take advantage of the acceleration provided by a grid using the VDB wrapper class which is derived from Volume<T>.

In writing images to disk, MeshPotato has the option to generate images in various formats since it uses OpenImageIO to handle the image conversion. MeshPotato can also generate deep images, which requires the OpenEXR format. A deep image is created during volume renderering since it uses the camera and distance along each ray that is marched to compute the multiple depths per pixel. The resulting deep images can later be used in other third-party applications like Nuke for deep image compositing.

[1] TBB, or Intel Threading Building Blocks, is a C++ template library for writing parallel C++ programs. More information is at http://www.threadingbuildingblocks.org



(a) Low Frequency Pyroclast

(b) High Frequency Pyroclast

Figure 4.3: Pyroclasts with different frequency parameters

The primary intent in creating this API is to be simple to use but yet offer a lot of control and flexibility to the artist in terms of modeling and rendering. Thus, parameters are made available for almost every class within MPVolume. For instance, it is possible to generate a variety of pyroclastic sphere styles using various noise parameters in the noise module, MPNoise (Figure 4.3).

Though the volume API has been written in C++, it also has a Python interface. This interface will be discussed in Chapter 5.

Chapter 5

Python Bindings

In production, it can be crucial to quickly build a tool to solve a particular problem. Scripting languages exist in third-party applications such as Maya, Nuke and Houdini to develop new tools to automate certain tasks. Python is a powerful scripting language and is also integrated in the previously mentioned applications as well as many others. Due to its portability and also the fact that Python is used extensively in the Digital Production Arts (DPA) pipeline[1], it is valuable for MeshPotato to have Python bindings. Boost.Python was chosen as the wrapper into Python in order to have control over the Python interface and because OpenVDB provides a set of Boost.Python classes for its Python module, pyopenvdb.

MeshPotato contains four Boost.Python wrapper classes, mpmesh, mpvolume, mpnoise and mputils. These classes parallel the organization of the C++ API. Exposing classes is relatively easy in Boost.Python. The classes member functions could have different names associated with them in Python. The caveat is that each class and each public variable must manually be exposed in order to be recognized in Python (Listing 5.1).

Listing 5.1: Exposing the Color class to Boost.Python

```
1 BOOST_PYTHON_MODULE(mputils) {
2     class_<MeshPotato::MPUtils::Color>("Color")
3                 .def(init<float, float, float, float>())
4                 .def("set", &MeshPotato::MPUtils::Color::set)
5         ;
6 }
```

[1]The DPA pipeline is a collection of tools and workflows that conform to a standard to faciliate the creation of animated films in the studio as part of the DPA graduate program at Clemson University.

Abstract base classes require an additional wrapper class to be defined. Once a Boost.Python wrapper has been completed, MeshPotato compiles the wrapper into a python module and creates a module. Storing multiple modules within a directory identifies them as being part of the MeshPotato Python interface and looks similar to namespaces in C++ (Listing 5.2).

Listing 5.2: Creating a Sphere in Python

```
1 # Creating a VDB levelset Sphere
2 import pyopenvdb as vdb
3 import pymeshpotato.mpvolume as mpvolume
4 import pymeshpotato.mputils as mputils
5
6 # Create A Sphere with radius 5 and center at origin
7 sphere = mpvolume.ImplicitSphere(5.0, mputils.getMPVec3([0,0,0]))
8
9 # Convert to VDB Fog Volume
10 minBB = mputils.getVDBCoord([-100, -100, -100])
11 maxBB = mputils.getVDBCoord([100, 100, 100])
12 vdb_sphere = mpvolume.makeVDBGrid(sphere,mputils.getVDBCoordBBox(minBB,maxBB),0.05)
```

Boost.Python also allows for the creation of functions that do not exist in the original C++ API. These functions can serve as a way to make the code behave more like the way Python is organized as opposed to the structure of C++. MeshPotato also uses functions to simplify the implementation details. Functions added to the module interface would read and export Python objects as parameters. The necessary data types can be extracted from the Python objects. Once extracted, they are treated as normal C++ variables. Finally to export, the variable is inserted into a new object and returned from the function (Listing 5.3).

Listing 5.3: Function example in Boost.Python wrapper interface

```
1 object ConvertToFogVolume(object &grid) {
2         openvdb::FloatGrid grid_obj = extract<openvdb::FloatGrid>(grid);
3         openvdb::tools::sdfToFogVolume<openvdb::FloatGrid>(grid_obj);
4
5         return object(grid_obj);
6
7 }
```

One important feature of the volume rendering API is that it must be able to take into account the aggressive memory cleanup of unreferenced objects in Python. For that reason, most routines in MeshPotato have an option to return reference-counted shared pointers. In Boost.Python, the default contructor is replaced with a call to the reference counted pointer object of the C++ class. This allows for the variables to behave more like Python variables. Reusing a variable will keep a reference to what the variable originally pointed to. This powerful feature, along with the

concept of resolution independent volumes, allows for the creation of complex volumes by simply using Python routines. Listing 5.4 demonstrates using a for loop in Python to create a ring of pyroclastic spheres (Figure 5.1).

Listing 5.4: Using reference counting variables to create complex volumes

```
1 def pyroLoop():
2         noise = mpnoise.Noise_t()
3         noise.amplitude = 0.1
4         height = drange(0.0, 6.5, 0.5)
5         sphere1 = mpvolume.PyroclasticSphere(0.01, mpvolume.getMPVec3([0,0.0,0]),noise)
6         for h in height:
7                 sphere2 = mpvolume.PyroclasticSphere(0.01, mpvolume.getMPVec3([math.cos(
                      h),math.sin(h),0.0]),noise)
8                 sphere1 = mpvolume.UnionFloat(sphere2, sphere1)
9         return sphere1
```

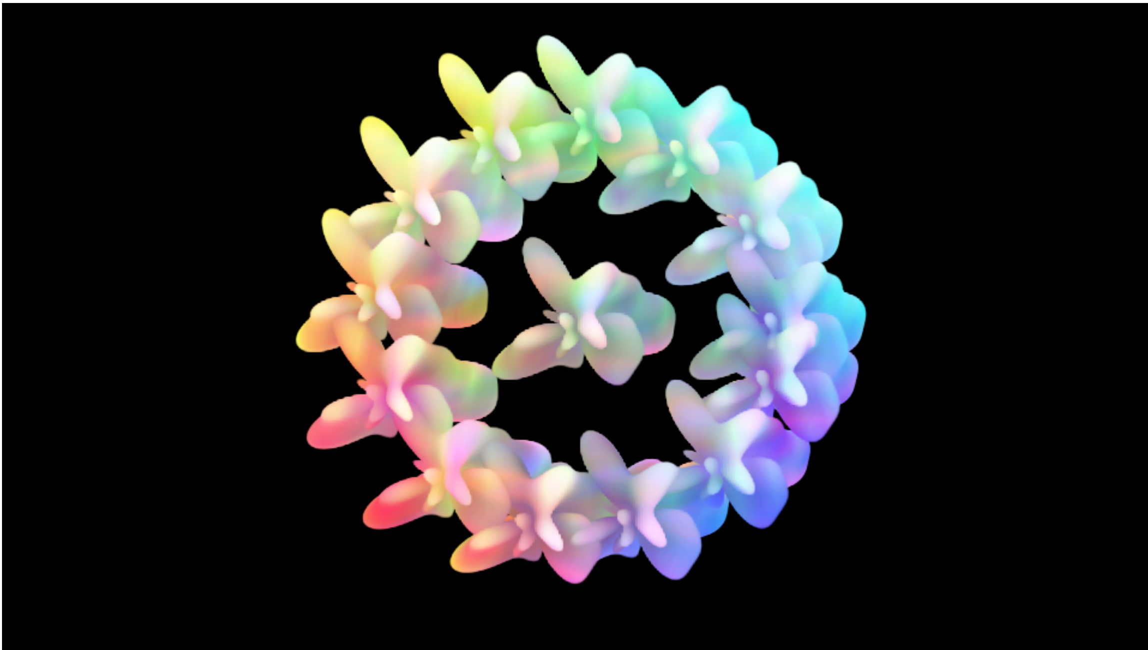


Figure 5.1: Rendered result from the pyroLoop function

Chapter 6

Results

MeshPotato is designed to be used in the creation of production tools to assist in the creation and visualization of volumes. MeshPotato also comes with several tools that interact within the command line and within Python. MeshViewer is a C++ program launched from the command line and is used to visualize mesh data. The tool was originally authored by Timothy Curtis, though it only supported Wavefront OBJ files. Using MeshPotato's plugin system, MeshViewer displays even more mesh types as long as a plugin is written for it. The advantage of this approach is that plugins can be dynamically authored and MeshViewer will be able to recognize them without changing its code base. Mesh information is stored in vertex buffer objects for realtime display in OpenGL.

MPConvert is able to convert polygonal geometry within Maya into a VDB levelset (Figure 6.1). Since it uses the Maya Python API, the process of obtaining the vertices, normals and faces of the geometry is relatively quick. This gives the artist the freedom to model geometry in a familiar toolset and convert them to levelsets without having to export the data into an intermediate file format such as a wavefront obj file (Figure 6.1). MPConvert has been used to assist in the modeling of clouds for DPA student Zhaoxin Ye's thesis, *Volumetric Cloud Rendering: An Animation of Clouds*. She created a simple base cloud model in Maya. Afterwards she launched MPConvert from the DPA Maya shelf, which provided options to control the voxel size and file path for the resulting mesh. Once the convert button was selected, a levelset would be generated and saved in the specified file path. Since VDB is a standard format, the levelset could be read into her cloud renderer (Figure 6.2).

MPVolumeRender is a command-line volume rendering tool that can handle VDB fog vol-

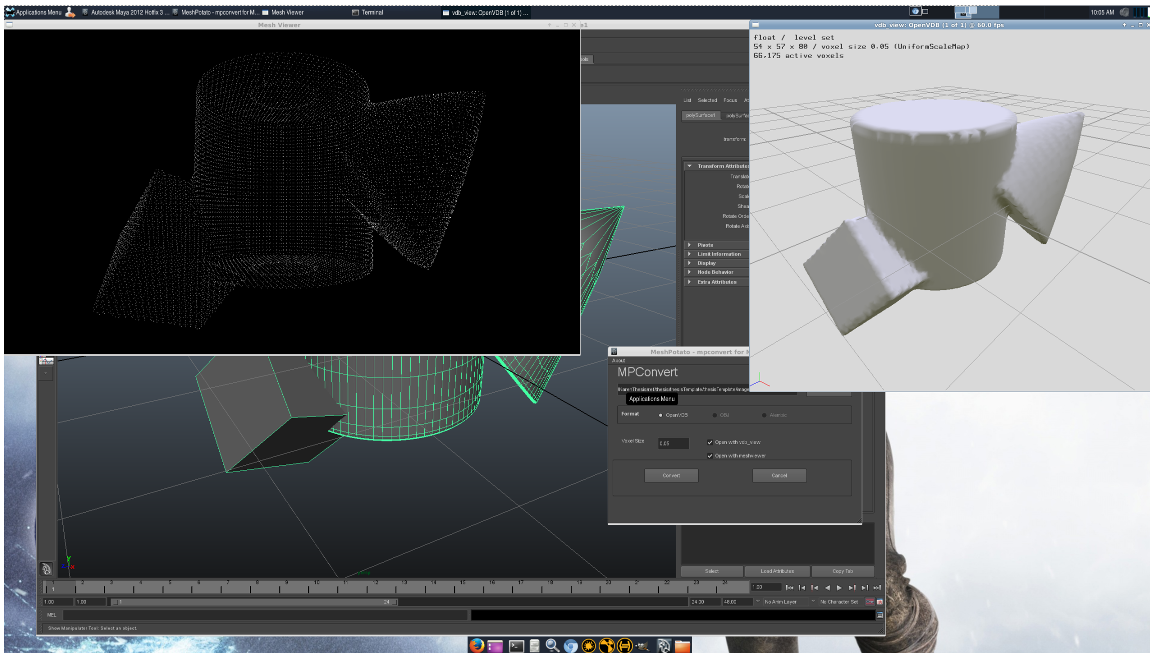
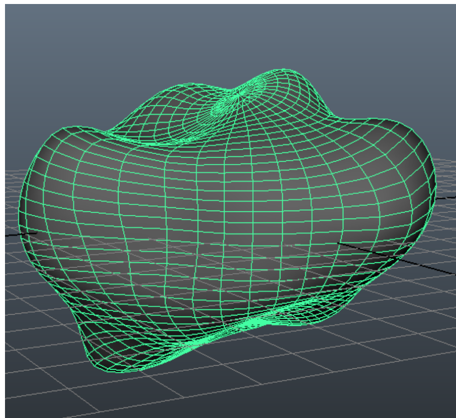


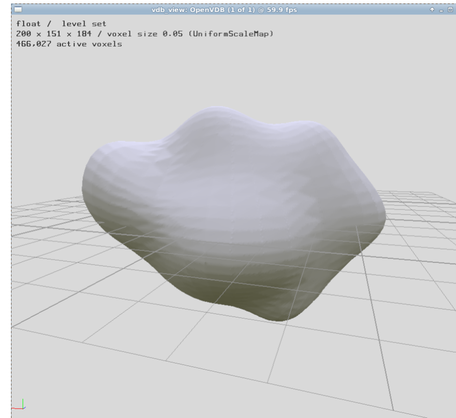
Figure 6.1: MPConvert for Maya, MeshViewer and vdb_view

umes. MPVolumeRender uses an RGB frustum light setup when computing DSMs for the fog volume. The ray marcher uses the OpenVDB Volume Ray Intersector, which makes the rendering process relatively quick. The fog volume of the public cloud bunny on the OpenVDB website can be rendered in about one and a half minutes on an Intel Xeon processor at 720p [1] (Figure 6.3).

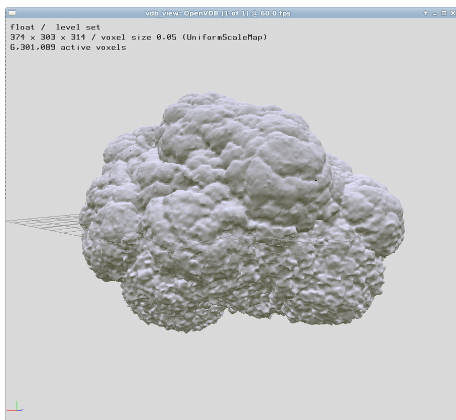
[1] The cloud bunny model can be downloaded from the OpenVDB website: http://www.openvdb.org/download



(a) Base Cloud Model in Maya



(b) Base Cloud Model converted to VDB levelset using MPConvert for Maya



(c) Cloud generated from Base Cloud Model in Zhaoxin's cloud renderer

(d) Final cloud image

Figure 6.2: Zhaoxin's Cloud Modeling workflow using MPConvert for Maya

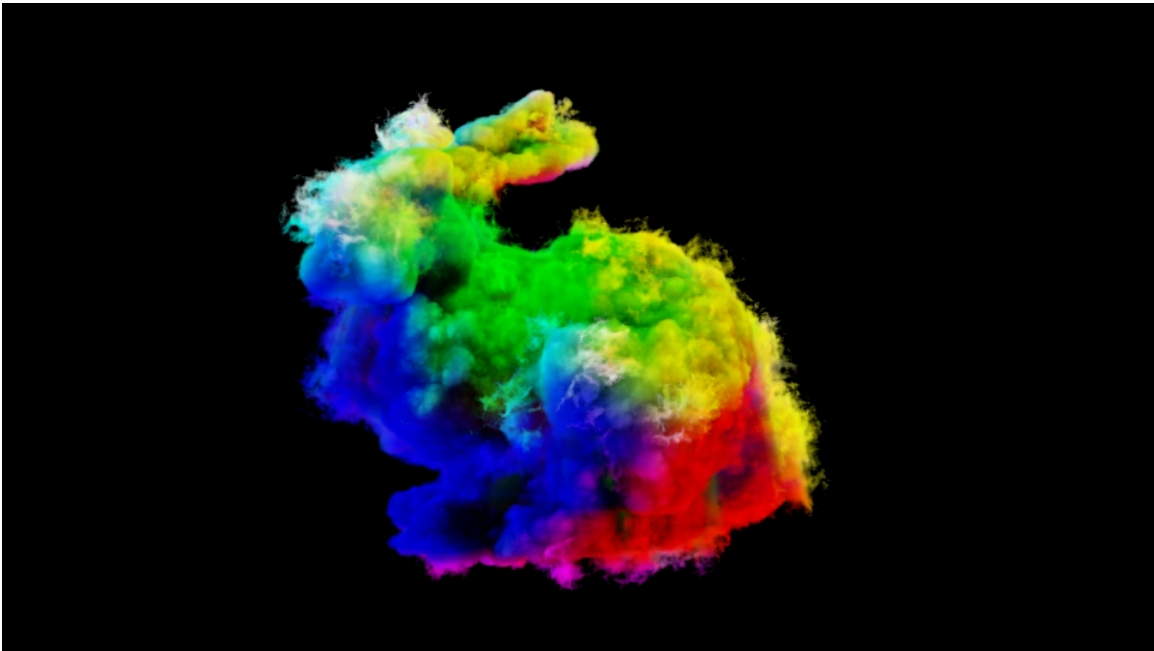


Figure 6.3: The OpenVDB public cloud bunny, rendered using MPVolumeRender.

Chapter 7

Conclusions and Discussion

MeshPotato provides an artist-oriented interface for creating high quality volumetric models and images. The Python bindings further extend MeshPotato by allowing the creation of tools which can be used with third-party applications such as Maya and Nuke or from within the command line.

MeshPotato has received several feature requests from individuals using it in the DPA studio. MeshViewer uses a default set of lights to view a model when shaded, though it can be convenient for the number of lights and their positions to be configurable by an artist so that the lighting can approximate what a render could potentially look like. This can also be used for planning the placement of lights within a volume to simulate internal lighting for phenomeneon such as clouds.

The design of MeshPotato should allow it to work within Houdini as a collection of SOP nodes. This interface is familiar with artists using Houdini and can introduce new functionality along side the existing set of nodes. OpenVDB provides CSG operations on volumes. It is possible to build a tool where models in Maya can be further manipulated with CSG operations. This can be convenient for modeling complex volumetric shapes. At times, it may be advantageous to work with subdivisions to reduce the polygon count. MeshPotato could be extended in the future with OpenSubDiv [8], where it can subdivide the input mesh data before converting it into a volume for smoother levelsets.

Appendix

The following code is an example of how a plugin can be written and read in by MeshPotato. This plugin comes with MeshPotato and allows for the conversion of mesh data into an OpenVDB levelset.

```
#include <MPPlugins/pluginapi.h>
#include <MPUtils/Vector.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <list>
using namespace std;
namespace MeshPotato {
namespace MPPlugins {
class VDBOutputMesh : public OutputMeshAPI {
public:
virtual ~VDBOutputMesh() {}
virtual const char *getName() const { return "VDB Output Mesh";}
virtual bool loadMesh(const list<std::vector<std::string> > &vertices,
            const list<std::vector<std::string> >&normals,
            const list<std::vector<std::string> > &faces,
            const MeshPotato::MeshSpec &spec) {
  std::cout <<"Loading VDB Output Mesh"<< std::endl;
  this->vertices.clear();
  this->normals.clear();
  this->faces.clear();
  this->vertices = vertices;
  this->normals = normals;
  this->faces = faces;
  voxelSize = spec.voxelSize;
  exBandWidth = spec.exBandWidth;
  inBandWidth = spec.inBandWidth;
  grid = openvdb::FloatGrid::create(/*background value */2.0);

  return true;

}
virtual bool writeMesh(const char *meshName) {
        std::stringstream strm;
MeshPotato::MPUtils::Verts verts;
MeshPotato::MPUtils::Polys polys;

// Write metadata
for (std::list<std::vector<std::string> >::iterator iter = vertices.begin();
 iter != vertices.end(); ++iter) {
float x,y,z;
strm << (*iter)[0];
strm >> x;
strm.str("");
strm.clear();
strm << (*iter)[1];
strm >> y;
strm.str("");
strm.clear();
strm << (*iter)[2];
strm >> z;
strm.str("");
strm.clear();
verts.push_back(openvdb::math::Vec3s(x, y, z));
}
```

Listing 1: VDBOutputPlugin.h (part 1)

```
58 // Write faces
59 for (std::list<std::vector<std::string>  >::iterator iter = faces.begin();
60 iter != faces.end(); ++iter) {
61 unsigned int x,y,z, w;
62 strm << (*iter)[0];
63 strm >> x;
64 strm.str("");
65 strm.clear();
66 strm << (*iter)[1];
67 strm >> y;
68 strm.str("");
69 strm.clear();
70 strm << (*iter)[2];
71 strm >> z;
72 strm.str("");
73 strm.clear();
74 if (iter->size() == 3)
75   polys.push_back(openvdb::math::Vec4<uint32_t>(x - 1, y - 1, z - 1,
76           openvdb::util::INVALID_IDX));
77 else if (iter->size() == 4) {
78   strm << (*iter)[3];
79   strm >> w;
80   strm.str("");
81   strm.clear();
82   polys.push_back(openvdb::math::Vec4<uint32_t>(x - 1, y - 1, z - 1, w - 1));
83
84 }
85
86 }
87
88
89 openvdb::math::Transform::Ptr transform = openvdb::math::Transform::
      createLinearTransform(voxelSize);
90
91 for( size_t i = 0; i < verts.size(); ++i ){
92 verts[i] = transform->worldToIndex(verts[i]);
93 }
94
95 openvdb::tools::MeshToVolume<openvdb::FloatGrid> levelset(transform);
96 levelset.convertToLevelSet(verts, polys, exBandWidth, inBandWidth);
97 // Export Mesh
98 openvdb::io::File file(meshName);
99 grid = levelset.distGridPtr();
100 openvdb::GridPtrVec grids;
101 grids.push_back(grid);
102 file.write(grids);
103 file.close();
104
105
106 return true;
107
108 }
109 virtual const unsigned int getNumberVertices() const {
110   return vertices.size();
111 }
112 virtual const unsigned int getNumberNormals() const {
113   return normals.size();
114 }
```

Listing 2: VDBOutputPlugin.h (part 2)

```
115 virtual const unsigned int getNumberFaces() const {
116   return faces.size();
117 }
118 virtual void setVoxelSize(float ) {
119   this->voxelSize = voxelSize;
120 }
121
122 private:
123 float voxelSize;
124 float exBandWidth;
125 float inBandWidth;
126 list<std::vector<std::string> > vertices;
127 list<std::vector<std::string> > normals;
128 list<std::vector<std::string> > faces;
129 openvdb::FloatGrid::Ptr grid;
130
131
132 };
133 PLUGIN_FUNC OutputMeshAPI *CreateOutputMesh () {
134 return new VDBOutputMesh;
135 }
136 PLUGIN_FUNC void DestroyOutputMesh(OutputMeshAPI *om) {
137 delete om;
138 }
139 PLUGIN_DISPLAY_NAME("VDB OutputMesh");
140 PLUGIN_INIT() {
141 std::cout << "PLUGIN_INIT_VDB_OUT" << std::endl;
142 RegisterOutputMesh("vdb", CreateOutputMesh, DestroyOutputMesh);
143 return 0;
144 }
145
146 }
147
148 }
```

Listing 3: VDBOutputPlugin.h (part 3)

Bibliography

[1] David Abrahams, U Koethe, RW Grosse-Kunstleve, et al. The boost python library. *Computer software. http://www. boost. org/libs/python*, 2002.

[2] David M Beazley et al. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[3] Peter Hillman. The theory of openexr deep samples. *http://www.openexr.com/documentation.html*, 2013.

[4] Florian Kainz. Interpreting openexr deep samples. *http://www.openexr.com/documentation.html*, 2013.

[5] Lee Lanier. *Professional digital compositing: essential tools and techniques.* John Wiley & Sons, 2009.

[6] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 385–392. ACM Press/Addison-Wesley Publishing Co., 2000.

[7] Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. Openvdb: an open-source data structure and toolkit for high-resolution volumes. In *ACM SIGGRAPH 2013 Courses*, page 19. ACM, 2013.

[8] Pixar. Opensubdiv. *http://graphics.pixar.com/opensubdiv/*, 2013.

[9] Martin Reddy. *API Design for C++.* Elsevier, 2011.

[10] Jerry Tessendorf. *Volume Modeling and Rendering.* 2013.

[11] Jerry Tessendorf and M Kowalski. Resolution independent volumes. *ACM SIGGRAPH 2010 Courses*, 2010.

[12] Magnus Wrenninge. Field3d: An open source file format for voxel data. *https://sites.google.com/site/field3d/*, 2009.

[13] Magnus Wrenninge. *Production Volume Rendering: Design and Implementation.* CRC Press, 2012.

[14] Magnus Wrenninge, N Bin Zafar, J Clifford, G Graham, D Penney, J Kontkanen, J Tessendorf, and A Clinton. Volumetric methods in visual effects. *SIGGRAPH 2010 Course Notes*, 2010.