APPROVAL SHEET

Title of Thesis: Polynomial Voxel Maps for Time Varying Volumetric Rendering

Name of Candidate: Kyle Boyer
Master of Science, 2018

Thesis and Abstract Approved: _____
Dr. Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

Title of Thesis: Polynomial Voxel Maps for Time Varying Volumetric Rendering

Kyle Boyer, Master of Science, 2018

Thesis directed by: Marc Olano, Associate Professor
Department of Computer Science and
Electrical Engineering

Volume visualization is used in a number of real-world applications, including scientific data representation, medical imaging, simulation display, and computer gaming. The standard method for rendering volumetric datasets in real time is Ray Marching. This is computationally expensive, and thus is almost exclusively offloaded to a discrete GPU. Modern graphics cards can handle the rendering costs, but the algorithm still does not account for time-varying datasets, in which the temporal changes in the data are relevant to the viewer. To handle pre-computed time-varying datasets, multiple snapshots of the data must be loaded to the GPU in real time, and interpolated. Datasets with many time steps can be too large to fit in GPU memory at once, but too slow to load one step at a time. I present a method to reduce or eliminate runtime loading, instead preprocessing all of the data instances before passing them to the GPU. I do so with a volumetric analog to polynomial texture maps, by fitting a polynomial function of time to the data as it changes in each voxel. These functions are then evaluated on the GPU at runtime. This approach allows handling of large time-varying datasets, while barely affecting rendering performance. The result is the ability to visualize, interact with, and explore large temporal volume data, in real time.

Polynomial Voxel Maps for Time Varying Volumetric Rendering

by

Kyle Boyer

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science - Computer Science
2018

Dedicated to Mom and Dad,

who kept me grounded here on Earth,

while encouraging me to fly among the stars.

ACKNOWLEDGMENTS

I would like to express my most sincere gratitude to my graduate advisor, Dr. Marc Olano, for everything he has done to help me in my career. I thank him for suggesting that I pursue Grad School, for the research opportunities he has brought me, and for consistently working with me and supporting me throughout my collegiate endeavors. I am thankful both for his immense experience and expertise in the domain of computer graphics, and for all of the skills and insights I have gained as a direct result of working with him. I am grateful for his words of wisdom and advice, and for all of his suggestions, feedback, and assistance during the completion of this thesis.

I would like to extend my sincere thanks to Dr. Neel Savani, for giving me the opportunity to pursue a research oriented position at NASA. Dr. Savani was pivotal in ensuring that I acquire and maintain the occupation that this thesis work came from. He has been especially invaluable, both to this thesis work, and to my career aspirations.

Finally, I would like to thank NASA Goddard Space Flight Center, the NASA Space Science Education Consortium, and the STEAM Innovation Lab. This includes Dr. C. Alex Young, Dr. Thomas Moore, Troy Cline, Bryan Stephenson, Troy King, Lani Sasser, and the rest of the STEAM Innovation Lab team. NASA was the primary funding source for this research work, and the STEAM Lab team under Dr. Young have been incredibly accommodating and supportive.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Visual representations of volume data are used in a multitude of professional applications, including scientific research, medical imaging, simulations, and game graphics. For all of these fields, differing requirements and types of visualizations are preferred. These vary from real-time, interactive volumes, to pre-rendered static displays, to Monte Carlo based particle systems. Each has their own set of benefits, but each also has specific constraints on rendering dynamics. The nuanced needs of these fields call for modifications to the way that volumes are displayed.

1.1 Applications

1.1.1 Games

In video games, it is quite common to need to display things that are not solid surfaces or objects. Fog, smoke, and clouds are all typical examples [Green 2005]. For most games, however, there is a tight constraint of 16.66 [Mark Claypool 2006], or 33.33 [Claypool and Claypool 2009] milliseconds of rendering time per frame, for a frame rate of 60 [Mark Claypool 2006] or 30 frames per second [Claypool and Claypool 2009]. For games in Virtual Reality, the frame rate increases to 90 FPS, which places a constraint of 11.1 milliseconds on each frame [Vlachos 2015]. Under these constraints, volumes are to be

rendered, and must look believable. If smoke is to be displayed, it should look like smoke to the player. Though, it does not need to draw its visualization from real smoke data, or simulations. If the player is standing in a field covered in a low fog, it needs to look like fog. It should obstruct light like fog [Wenzel 2006], and it could even move as the player traverses through it [Müller et al. 2003], but it does not need to represent some physical fog dataset.

This means that data does not need to be loaded from memory, or passed to the processor handling the rendering, for the computation to take place. Instead, a phenomenological method will be evaluated to simulate what smoke from the chimney might look like, which removes the overhead of loading a large volumetric dataset to sample. The concept of a systematic runtime evaluation to give expected results is borrowed from game volumetrics in this thesis, and carried over to scientific data volumes. This is a useful notion, and when applied to real datasets, can improve performance dramatically.

1.1.2 Medical Data

For medical imaging, authentic data must be used. Visual accuracy is of paramount importance, as the data to be displayed is directly collected from patients, and is used for diagnosis and tracking [Smelyanskiy et al. 2009]. The classic medical example is a Magnetic Resonance Imaging scan, or MRI. An MRI is an inherently volumetric piece of data, that represents the spatial density of physical matter. The requirement of using real data, such as an MRI, necessitates loading that data. For a visualization of a subject to be possible, the volumetric information representing that subject must be accessible to the processing unit doing the rendering. In the context of computer hardware, this means that memory bandwidth is now a factor that affects performance.

1.1.3 Scientific Data

For scientific data, which is the primary focus of this thesis, data loading is also required. Additionally, the ability to explore and analyze the data in whatever way the user desires is essential. If the user wishes to rotate the visualization, zoom in, change the color scaling, or view slice planes, they must be able to do so. To accommodate this requirement, a real-time approach is used, just like in a game. For the purpose of this thesis, Virtual Reality will be used, for the immersive experience it offers the user viewing the data. So, the constraint of 11.1 milliseconds per frame is in effect, along with the data loading demand.

Scientific data often models naturally occurring phenomena, such as magnetic fields [Takahashi and Anderson 1992], or solar wind [Lundstedt and Wintoft 1994]. To effectively visualize this type of data, a rendering of a single static volume is sometimes insufficient [Woodring and Shen 2003]. Instead, a time-varying visualization is preferred. To accomplish this, multiple snapshots of the subject data are used, and cycled through in sequence. When this is done in real time, the user is able to see the natural transformation of the volume over the series of volumetric timesteps.

1.2 Computational Complexity

Handling temporal volumes in real-time is computationally expensive. In addition to the per-frame rendering of the volume itself, transitioning between timesteps introduces its own problems. In the current method for calculating pixel color for volumetric displays, multiple timesteps of the data must be loaded into memory. [Decker 2007] At a minimum, two timesteps at once must always be present in memory, the current timestep, and the one directly following it. We will see in the following section that three timesteps can, and should be used.

The loading and unloading of volume snapshots from active memory creates a data

load slowdown. The rendering must wait for the load to be completed before accessing the information in memory for sampling. The amount of memory required is also increased significantly, to store multiple steps concurrently. As a contextual example, a dataset with a volume size of 512x512x512 points would be 512 megabytes alone, but would require 512 * 3 = 1536 megabytes for multiple steps. The dataset in the Visible Human Project is 15 gigabytes [Fudenberg and Kreps 2018], so 45 gigabytes of memory would be required, of which 15 gigabytes would need to be loaded at runtime.

In this thesis I introduce a methodology to remove (or reduce) the data loading: Polynomial Voxel Mapping (PVM). I will discuss the algorithmic details and benefits, the implementation, and the measured results of this technique. If this procedure is adapted and utilized, time variant datasets can be displayed almost as easily and perform almost as well as purely static datasets.

The rest of this document is laid out as follows: Chapter 2 discusses related work, what has been done in the past, and background information relevant to the algorithm. Chapter 3 describes the method itself, in detail. Chapter 4 is an in-depth presentation of performance results. The final chapter includes closing thoughts, and potential future work.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Volume Rendering

There are a multitude of strategies that have been explored for rendering volumes of data to a computer screen. These include algorithms like splatting [Westover 1991], shear warp [Lacroute and Levoy 1994], and the most commonly used one in modern graphics — volumetric ray casting. [Levoy 1988]. This thesis will use the standard approach of volumetric ray casting to render the data from a block of information to a picture on the screen. Ray casting (also known as ray marching) operates by calculating and firing a virtual ray that originates at the viewer's eye, and travels through a particular pixel on the viewing plane. Each pixel on the screen has its own individual ray. Thus, all travel at slightly different trajectories. The ray, after being fired, travels along the vector, and passes through the volume to be visualized [Levoy 1988]. This is similar to the *ray tracing* technique for rendering surfaces [Appel 1968].

Where ray marching differs from ray tracing is how the calculations are done along the ray itself. In ray tracing, the ray is checked for an intersection point with the surfaces in the scene. Once all of the surfaces have been checked, the closest intersection point to the eye is used, and the ray then returns a color based on the surface hit by that intersection. So, the surface in the front at any particular pixel obstructs the surfaces behind it. This concept

is illustrated in Fig. 2.1

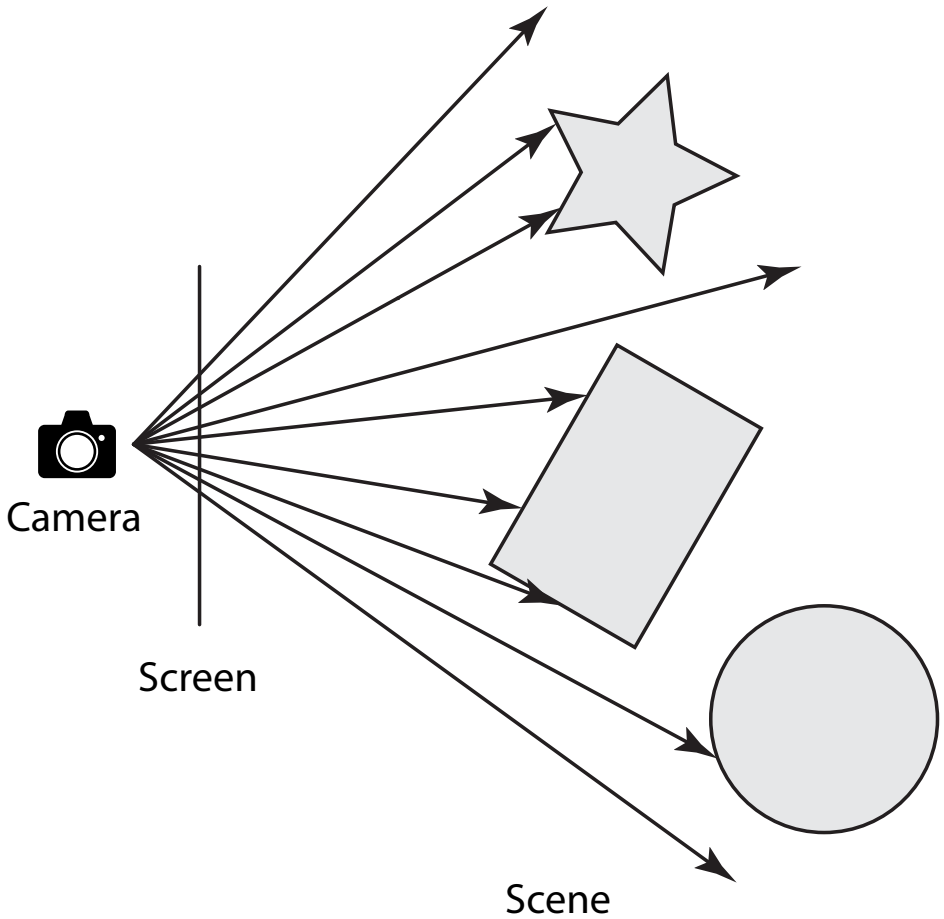


FIG. 2.1. Basic ray tracing algorithm

In ray marching, the rays hit the data volume, rather than polygonal surfaces. Contained within the volume are the data points of interest, to be visualized. Each point represents a position in space on a uniform 3D grid. These points are called voxels [Foley

et al. 1990]. When the fired rays intersect with the outer shell of the volume, they begin to repeatedly sample into the voxels. Every time a sample has been taken, the ray progresses a set distance, and another sample is done. This continues until the ray has passed all the way through the volume. An illustration of this process is shown in Fig.2.2.

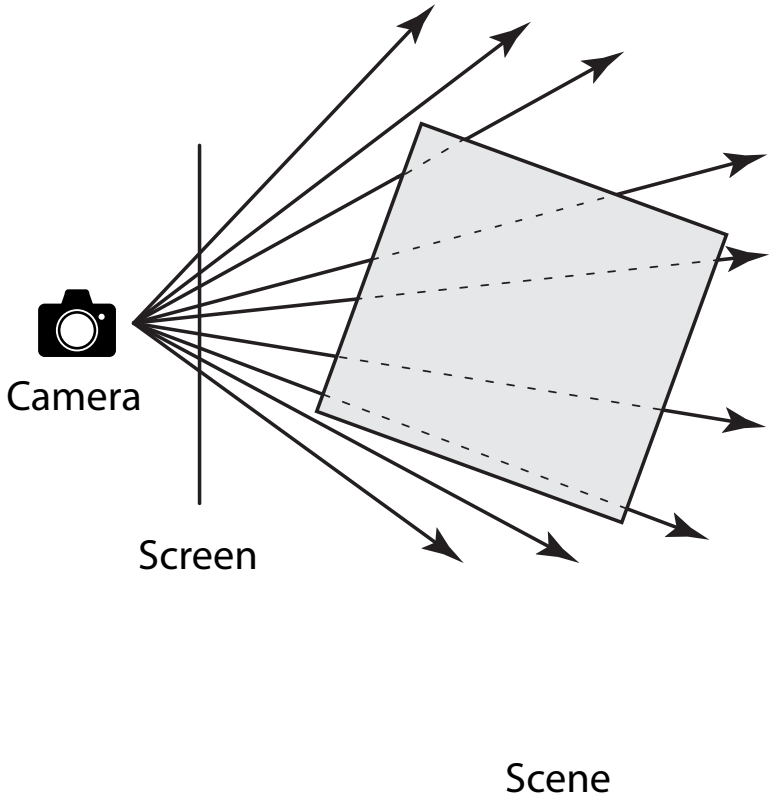


FIG. 2.2. Basic ray casting algorithm

In this way, every ray contains an aggregate value that represents all (or at least most) of the voxels that it has intersected with along its traversal path. When this process is completed for all of the pixels on the screen, the output image is a projection of the volume on to the viewing plane.

2.2 Time Varying Data

Expanding volumetric ray marching to be able to handle data that varies over time changes the requirements somewhat dramatically. Time varying data is often generated and stored as regular snapshots of a subject of interest [Lu and Shen 2008]. To be able to show the temporal transformation, these snapshots must be displayed to the user in sequence. Still, it is not enough to simply cycle the instances of data one by one. This would result in abrupt, sharp changes that resemble a slideshow of the volumes. Instead, the volume should look 'animated.' It should appear to flow naturally, as it would in the real world. To accomplish this, there must be a way to determine what the volume will look like in between the snapshots of the dataset, so that as the rendering is executed, each frame displayed to the user has a unique image.

2.2.1 Interpolation Approach

A few different methods have been explored to achieve realistic looking volumes that 'flow'. Some of these build on the basic approach to time-varying volume rendering, which is to find substep voxel values by interpolation [Voigt 1998]. The goal of time-varying volume visualization is to be able to find acceptable data values for any point between two given volume snapshots. A straightforward way to do this is to use the two snapshots on either side of the time in question to interpolate the time itself. For example, given a timestep $t = 1$ and another timestep $t = 2$, say we wish to find the value of the data at

$t = 1.4$. A simple linear interpolation will give the result we want. While this method is simple to understand and implement, it has some performance limitations that restrict its use.

Linear interpolation takes three input variables: the first point, the second point, and the interpolation coefficient. In the case of time varying data, the interpolation coefficient is the current time. The first and second data points are from the input voxels. This raises a problem. To be able to linearly interpolate between two voxels, both voxels must be loaded into memory for use. Consequently, to be able to interpolate every voxel within a volume of data, all voxels for both volumes must be in memory at once. Therefore, for large datasets, memory capacity can be restrictive.

Modern implementations of ray marching offload the rendering computation to a discrete GPU for performance. Discrete GPUs contain their own memory, which is used for computation on the GPU. This specialized GPU memory is often called VRAM (Video RAM). On modern graphics hardware, the memory is roughly 2-4 gigabytes in size [nVidia 2018]. Offloading the computation requires the voxel data to be loaded to GPU memory for sampling. When interpolating between two volumes, both must be sent to the GPU. Initially, this poses no problems, as both volumes can be passed at program start, and then sampled as the application progresses. The issue arises when the t value reaches the second volume. If there are subsequent volumes to be visualized, they too must now be loaded into GPU memory. This means they have to be read from disk, loaded into CPU memory, and then passed off to GPU memory, all during execution. To avoid this, all of the volumes could be loaded to the GPU at once, at program start, but in that case, the GPU memory buffer must be large enough to hold all of the timesteps concurrently. This is not feasible on many GPUs when working with large datasets, due to the limited capacity of the GPU memory.

The major problem with runtime loads is that the execution of the rendering stops

entirely while waiting for the data to be loaded. When it is time to load a new snapshot of data to interpolate, the renderer must wait until the load is complete before continuing, because it needs that data to execute the interpolation routine. So, the program stalls. For a real-time rendered application, this is unacceptable.

A possible way to alleviate the stall that occurs when loading new data at runtime is to load a block of data while interpolating different ones. The third volume of data in a series can be loaded to the GPU while the first two are being interpolated. Then, when the interpolation is complete (meaning the interpolation coefficient has reached 1.0), the interpolation restarts. When the next snapshot in the sequence is ready to be rendered, it is already in memory. The second and third volume then get interpolated, while the fourth one is loaded, and so on. So, while interpolating $t = 1$ and $t = 2$, we load $t = 3$ into memory. Decker [2007] used this approach for loading time-varying vector fields for flow visualization.

This method is not without flaws. It requires three volumes to be concurrently loaded into memory at any one time — the first one being interpolated, the second one being interpolated, and the one being loaded for interpolation later. If all three of these do not fit into VRAM together, which is a possibility for large datasets, this approach will not work. Additionally, stalls can still occur if the volumes are cycled too quickly. If the amount of time needed to load a new volume is greater than the amount of time allocated to interpolating between the ones already loaded, a stall for the remaining load time will still happen. In order to prevent this, the data must be cycled slowly enough that there is time to load an entire volume, and pass it to the GPU, during a single interpolation.

2.3 Polynomial Texture Mapping

This thesis proposes a solution to the problems of interpolating time varying data, by adapting ideas from a technique known as Polynomial Texture Mapping. Polynomial Texture Mapping was introduced by Malzbender et al. [2001]. The idea behind Polynomial Texture Maps is to create textures that store polynomial functions in each texel, rather than color data. Then, when these textures are applied to a surface, the renderer uses the functions stored at each texel to alter the appearance of the surface, based on some input parameter.

The primary usage in the Polynomial Texture Mapping publication is to simulate light hitting objects. A typical texture for a surface stores diffuse color data. When applying the bidirectional reflectance distribution function (BRDF), the color of the surface is sampled from the texture. The surface is then lit according to the BRDF. Often, a normal map (bump map) is applied to the texture as well, to perturb the surface normals of the otherwise flat surface to make it appear as if it has finer detail. Instead, when applying a Polynomial Texture Map, the incoming angle of light is used as the input to the polynomial function of the light direction stored at each texel. Then, when that function is evaluated, it produces a color value that is applied to the surface.

To build the functions, Malzbender et al. [2001] took photographs from a fixed position, with light coming from varying angles. From these pictures, they constructed polynomial equations that mapped the lighting directions to the expected pixel colors. Once they had a polynomial for every pixel, they constructed a texture containing the coefficients.

The work also discussed the viability of using other input parameters in place of lighting angle. Specifically, they demonstrated examples fitting a polynomial function to both depth of focus and time. Given pictures of a scene with differing levels of focus, the maps could be created to render that scene with any level of focus. The same is true for a scene

at different times. This was the inspiration for Polynomial Voxel Mapping.

In this thesis, I will discuss the use of Polynomial Voxel Mapping, to render time varying scientific data. My input will be a temporal series of volumes, extending the Polynomial Texture Mapping approach to visualize 3D science data in real time.

Chapter 3

METHODOLOGY

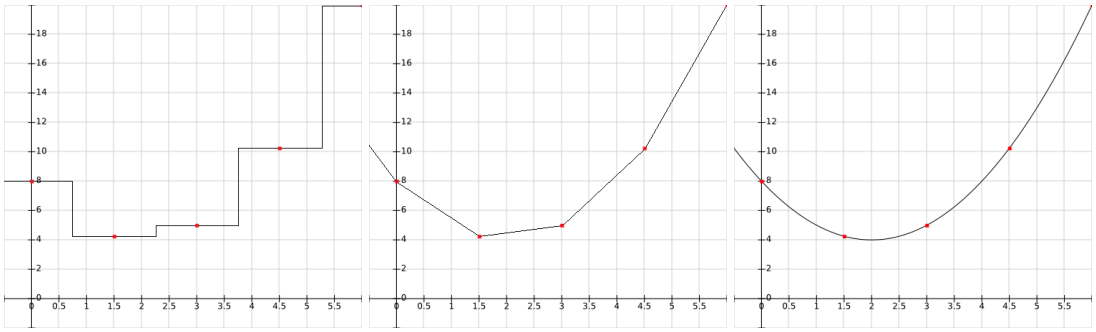
3.1 Approach

This thesis introduces Polynomial Voxel Mapping as a technique for handling large volumetric datasets, that contain many static temporal 'snapshots' over a potentially long period of time. Each snapshot, or time step, of the dataset is effectively an entirely new collection of data, that must be handled individually. Thus, for a data series with two snapshots, there is twice as much data to handle as a static dataset. With three, it becomes triple, and so on. Handling that much memory is bandwidth and capacity intensive. Also, each snapshot is not necessarily a smooth transition from the previous one. The instances could be sparse, with long gaps in between them. So, to display the data effectively, the values for every frame between two instances must be estimated.

3.1.1 Algorithm Overview

Polynomial Voxel Mapping attempts to solve both of these problems by providing a way to estimate per-voxel values of a dataset. Instead of using the raw voxel information from the input sets as the data values for intermediate frames, the voxels will be mapped to a polynomial function. The input sets represent exact values for a particular time value. Flipping through the timesteps shows only the exact data, but is piecewise constant. The

interpolation approach is piecewise linear. The PVM method is, instead, a higher order interpolation that uses more of the data points to determine a smooth curve. This difference is shown conceptually in the figure below.



(a) Piecewise Constant (No transition)

(b) Piecewise Linear (Interpolated transition)

(c) Function Fitted (PVM transition)

FIG. 3.1. Three different methods of finding substep timings of voxel values.

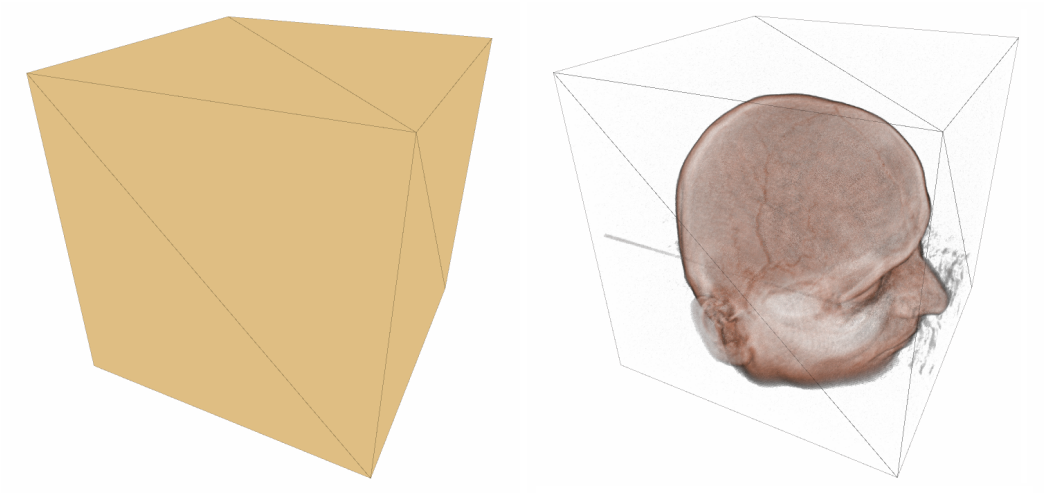
Once the polynomial function representation has been found for every voxel, the functions will be evaluated at runtime for every frame of execution, using the current time as the input. Rather than a linear interpolation between two points, the expected value of the voxel will be a point along a smooth polynomial curve. Thus, every per-frame, per-voxel output will be an approximation of that voxel's value, and can be predicted for any frame without needing to load the raw data. Instead, a functional representation of each data point is passed to the renderer. Since this does not change at runtime for small numbers of timesteps, such as the data used in this work, it can be passed once at program start, rather than streamed during execution.

3.2 Implementation

3.2.1 Volumetric Ray Marching

The first step in implementing the Polynomial Voxel Mapping approach is to develop a standard real-time volumetric raymarcher, to serve as the basis for the algorithmic modification. This ray marching method is the industry standard for real-time static volumes [Levoy 1988], and is adapted to an interpolation based technique to accommodate time-varying data. This interpolation procedure will be entirely removed and replaced with the PVM implementation.

As is the standard in real-time volume rendering, the entirety of the ray marching implementation will be offloaded to the discrete graphics processor. This facilitates a vast performance increase over running the code on the CPU, and additionally frees up the CPU to handle data management, and other potential needs that the program utilizing the volumetric display may require. The parallel nature of the GPU allows many pixels to be calculated simultaneously. So, the ray marcher, with all of the modifications for Polynomial Voxel Mapping, should and will be implemented to run on the GPU. For my implementation, I used the Unity game engine, which implements Microsoft's DirectX, and developed the renderer in High Level Shading Language (HLSL). Once the volumetric renderer is created, it can then be applied to a piece of simple proxy geometry. This geometry's shape will mirror that of the data volume itself. In the case of my test environment, a simple cube. Then, when the code is executed, the HLSL ray marcher will run for each visible pixel of the cube. The geometry surface will essentially act as a 'window' to the volume data. This is shown in Fig. 3.2.



(a) Cube shaded with solid color. (b) Cube shaded with volume shader.

FIG. 3.2. The same cube, colored differently.

To create the time-varying volume renderer, the first step is to implement standard ray marching. See the pseudocode in Lst. 3.1.

```
// simple volume rendering ray marcher

#define MAXSTEPS 150          // max samples

// textures and parameters
Texture2D colorMap;           // Color lookup texture
Texture3D volumeData;         // Volume texture data
float stepSize;               // Length of one step
int currentFrame;             // Frame count

// info passed from vertex shader
float4 camPos;                // eye location in object space
float4 pos;                   // position in object space

void main()
{
    float3 rayDir = normalize(pos - camPos);
    float3 stepVec = rayDir * stepSize;
    float3 exitPoint = calcRayExit();
    float totalDistance = objectPos - exitPoint;
```

```
21
22      // marching loop
23      Color result = float4(0.0);
24      float s=0.0;
25      for(int i=0; i<MAXSTEPS && s<=totalDistance; i++)
26      {
27        float4 voxelData = sample3D(volumeData, pos);
28        float4 color = sample2D(colorMap, voxelData);
29        result = result + (1.0 - result.a)*color;
30        s+=stepSize;
31        pos+=stepVec;
32      }
33      return result;
34  }
```

Listing 3.1. Ray marching pseudocode

At the beginning of the pseudocode, we see the important variables for the algorithm. `MAXSTEPS` is the maximum amount of samples along a single ray that the algorithm will take. Below that, two texture variables are listed. $volumeData$ is the 3D grid of voxel data, from the input. The other texture, $colorMap$ is a special piece of information that will be used during the marching loop. Whenever a sample is taken into the volume data, the result of that sample needs to be converted into a color that the viewer's eye can see. This texture serves as a lookup table to find that color.

Before the sampling can begin, the 3D vector representation of the ray to be marched must be calculated. This is the ray starting at the viewer's eye (the camera) and pointing through the pixel to be computed. The ray can be found by

$$rayDirection = locationOnGeometrySurface - cameraLocation \tag{3.1}$$

As seen on line 17 of Fig. 3.1. Both of these values are readily available through DirectX's API, given that the proper rotation matrices have been passed through as accessible variables. Game engines such as Unity and Unreal Engine 4 provide these matrices for GPU

code. Once the ray is found, the exit point on the geometry can be found (line 19), and the total length of the sampling line is

$$rayTravelDistance = |ExitLocation - EnterLocation| \tag{3.2}$$

as in line 20. This total length value will dictate the exit condition for the ray march. If we have marched farther than the total distance from start to finish, the algorithm may terminate safely.

The marching loop is now set up. The march will begin at the surface of the geometry, and each loop iteration will progress along the ray vector by a length of $stepSize$. For each execution of the loop, a sample will be taken into the volume data, at the 3D position of the ray step (line 27). The return value of that sampling will be used to do another sampling, into the color map (line 28). The result of that will be a visible color that can be added to the ray's cumulative color total. Next, that color value needs to be scaled by the accumulated opacity thus far through the volume. (line 29). Then, lines 30 and 31 increment the distance traveled, and progress one step forward along the ray. Once the march has reached $MAXSTEPS$ iterations, or the distance traveled is greater than $totalDistance$, the loop ends.

The smaller the $stepSize$, and the larger the $MAXSTEPS$, the more samples the renderer will take. More samples will give a more accurate result, but every sample taken is more work the GPU has to do. Thusly, the more accurate the result is, the less performant the renderer will be. Now, the only thing left is where to get the data to sample. Normally, as in this pseudocode snippet, the raw voxel data is used. Instead, for Polynomial Voxel Mapping, the volume data must be constructed from the raw data during precomputation.

3.2.2 Functional Voxel Representation

For a static volume, the data itself can be packaged up and sent to the graphics hardware. As long as it can be moved from standard CPU memory to VRAM, the hardware can access it directly for sampling. This requires no extra computation. However, to support time-varying data, we preprocess it using the PVM algorithm before handing it over to the GPU. Rather than passing raw data, we will pass a representation of the data where each voxel contains four coefficients. These coefficient values are floating point real numbers that construct a cubic polynomial.

The function that includes these coefficients uses the standard polynomial form

$$f(t) = At^3 + Bt^2 + Ct + D \tag{3.3}$$

Using the Polynomial Voxel Mapping approach, the set of all of these cubic polynomials represents the temporal transition of the volume. The change over time of the voxels will be mapped to the cubic function's output. Each voxel will be mapped individually, as each has their own unique potential change. In this way, a smooth curve for every voxel of information can be constructed, rather than a direct, 'staggered' transition. The difference is shown in Fig. 3.3

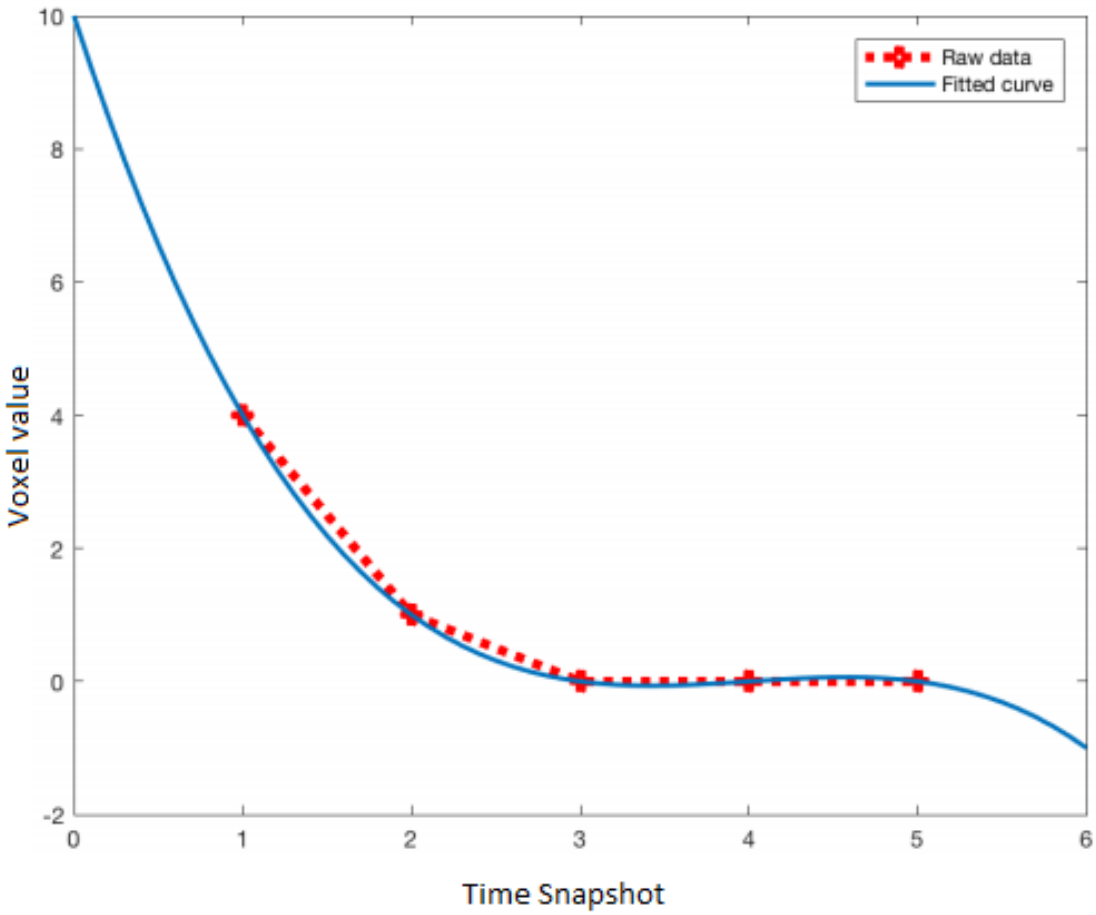


FIG. 3.3. Linear interpolation of data, vs. fitted curve

Since every voxel contains a unique function, specific to its own data change, the final result of this approach will be a uniform 3D grid of polynomials. The dimensions of this grid will match the dimensions of the input volume data. Meaning

$$VolumeDimensions(x, y, z) = PvmDimensions(x, y, z) \tag{3.4}$$

This also means that every voxel must be computed individually. Since all of the computation will be preprocessed, this does not affect the final running time.

In equation 3.3, $f(t)$ is the output, and represents the expected voxel value at a given t time. t is the input variable, representing current time, which will be extracted and passed from the frame counter, on a per-frame basis. The set $[A, B, C, D]$ represents the coefficients that make up the polynomial, and dictate the shape of the curve. These coefficients form the backbone of the PVM algorithm. Once they are obtained, the completion of the algorithm becomes trivial — evaluate a third degree polynomial. So, we must find these coefficient values. To calculate a polynomial function that best fits a set of data, we apply regression analysis. Namely, multivariable linear regression.

3.2.3 Regression Analysis

To map a dataset to a function using linear regression, an estimation method must be chosen. For Polynomial Voxel Mapping, we use Ordinary Least Squares (OLS). OLS is the most commonly used estimation technique for linear regression, and one of the most straightforward to understand.

The function starts in the form $f(t) = At^3 + Bt^2 + Ct + D$. This can be converted into matrix form for the process of OLS. To do so, we treat the coefficients and the known function outputs as vectors. Thus we have the vector β of coefficients: $\beta = [A, B, C, D]^T$, and the vector Y of known output values: $Y = [y_1..y_n]^T$. In my test dataset, there were five input timesteps to map, so we have $Y = [y_1, y_2, y_3, y_4, y_5]^T$. These values in the Y vector represent the contents of each of the sequence of input voxels, for a certain grid position. They also represent the expected output of the function $f(t)$, for all 5 known values of t, $[t_1..t_5]$.

We now can rewrite the polynomial equation from standard form $f(t) = At^3 + Bt^2 + Ct + D$ to matrix form

$$Y = X\beta \tag{3.5}$$

where X is the input matrix of variables. X is constructed from $[t_1..t_n]$ and the corresponding exponential multiples of t, $[t^0..t^3]$. The dimensions of X will be four columns by n rows, where n is the number of input timesteps. For my test dataset, I had $[t_1..t_5] = [1, 2, 3, 4, 5]$. Thus, the X matrix was:

$$X = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ 1 & 4 & 16 & 64 \\ 1 & 5 & 25 & 125 \end{bmatrix} \tag{3.6}$$

It is worth noting that the input timesteps need not be evenly spaced. For my data, There was a step length of 1 between every t value, but something like $t = [1, 2, 6, 8, 15]$ is acceptable, and will work the same.

Now that the equation is in matrix form, it can be solved for β using Ordinary Least Squares. Thus we have

$$\beta = (X^T X)^{-1} X^T Y \tag{3.7}$$

By computing this equation for β, using standard matrix multiply operations, we have successfully found the four coefficients $[A, B, C, D]$. Now, by using these coefficients in the original standard form of the equation $f(t) = At^3 + Bt^2 + Ct + D$, we can successfully evaluate the equation for any value of t, rather than just the $[t_1..t_n]$ that we started with. This means it is now possible to find a new value of $f(t)$ at every frame.

3.2.4 Volume Construction

The set of coefficients must be calculated and stored for every unique voxel. This requires $(x*y*z)$ computations. This can be done in parallel, if desired, but it will not affect

eventual running time, once the preprocessing is complete. So, for ease of implementation, my computation was done serially. As the coefficients are found, they should be stored in memory, in a 3D grid based data structure.

Once the entire volume of data has been processed, and the grid of all β coefficients has been created, it needs to be passed to the graphics hardware, so that the ray marcher can access it. This is where the pre-existing capabilities of the graphics API prove useful, both for ease of use, and performance. DirectX and HLSL support texture sampling as a standard, core feature. This built-in process involves sending an array of color values to the graphics memory, and then performing memory lookups into the array, to check the color at a particular index or to interpolate between values in the texture grid. The colors are up to 4-channel (RGBA) floating point vector values, and the array is indexed into like a grid, using 0-1 values in each of the axes. In addition to typical 2D texturing, GPUs support 3D textures, which are essentially like stacked layers of 2D textures. Coupled with x and y, they have a z index component. This lends itself nicely to volume data.

Rather than develop a new method for passing the PVM data to the graphics card, we will use what is already there. The 3D texture functionality of the graphics API is perfect for our needs. It already handles loading, and sampling, and it is thoroughly optimized to take advantage of the GPU's parallelization. By accessing the polynomial coefficients that were calculated from the regression, and treating them as color channel variables, we can construct a texture. We will access a single element of the PVM volume constructed previously, and convert it to an RGBA color variable. We change the vector of coefficients $\beta = [A, B, C, D]$ to a vector of color intensities $Col = [R, G, B, A]$. So, A becomes Red, B becomes Green, C becomes Blue, and D becomes Alpha. We will then store this new Color vector in a 3D array of colors. Once all of the new color values have been created, we treat them all as texels, and write a Tex3D of RGBA32 values. We now have a volume that can be passed to the GPU.

Now that the data is accessible on the hardware, samples can be taken. To do so, the 3D space within the volume is used as the sampling coordinates for the 3D texture. At every sampling location, a voxel will exist containing the four coefficients that were passed as color values. So, when performing a texture lookup, the return value will be the 4-item vector, $[A, B, C, D]$.

This is the vector β that resulted from the original regression mapping. The values are the coefficients in standard cubic polynomial form. Where a typical texture lookup would have returned an immediately usable value, this vector can be used to find the value we want. The cubic polynomial must be evaluated to retrieve the volume quantity. In the test data set, the quantity represents density at the voxel. So, we have $f(t) = voxelDensity = At^3 + Bt^2 + Ct + D$, where $[A, B, C, D]$ is the result of the texture sampling. t is then the input variable, and represents the current time. This is just the current frame. By passing the frame counter for t, we can compute any value of $f(t)$ in real time, individually per voxel.

3.2.5 Renderer Modification

Adding a simple function to evaluate a cubic polynomial with the given input coefficients and t value, as shown in Lst 3.2 allows reconstruction of the PVM, per frame.

```
27        float4 voxelData = sample3D(volumeData, pos);
28        float tval = polynomial(voxelData, currentFrame);
29        float4 color = sample2D(colorMap, tval);
```

Listing 3.2. Updated Ray marcher

This polynomial will return the expected volume data value at any input time. On line 28 of Lst. 3.2, this polynomial is seen. It takes $voxelData$ as the input, which was sampled from the volume. It uses the coefficients stored in $voxelData$, and the current execution frame to evaluate a polynomial function. The result of the polynomial is then mapped to an

output color on line 29.

Once the ray march has concluded, and the execution loop terminates, the final pixel color has been found. We are left with a single 4-channel RGBA color value that is output from the graphics hardware as a color on the computer screen. This is line 32 in the original pseudocode (Lst. 3.1). With this additional modification, it is now possible to process multiple timesteps without needing to load them each to VRAM individually.

Chapter 4

RESULTS

4.1 Dataset Information

4.1.1 Background

The results for my implementation were all created with data generated by NASA Goddard Space Flight Center's Community Coordinated Modeling Center. Henceforth referred to as the NASA CCMC. The CCMC is a NASA entity that coordinates and supports research for space science and space weather modeling on behalf of domestic and international organizations. They provide data, forecasting information, and models to the science community within NASA. [NASA 2018a]

The dataset used for development, implementation, and testing of this thesis was created by the CCMC, using simulation software developed by the University of Michigan, entitled EEGGL. [Jin et al. 2017]. The data respresents the progression of a Coronal Mass Ejection, or CME. A CME is an eruption of plasma and magnetic energy from the corona surrounding our Sun [Gleber 2014]. They typically coincide with solar flares, and the plasma released goes directly into the solar wind.

CMEs travel outward from the sun, and are large and fast enough to actually collide with the Earth. A CME will usually hit Earth in 2-4 days after its ejection from the Sun's corona, as it travels at typically 500 - 1500 kilometers per second [Gleber 2014]. The

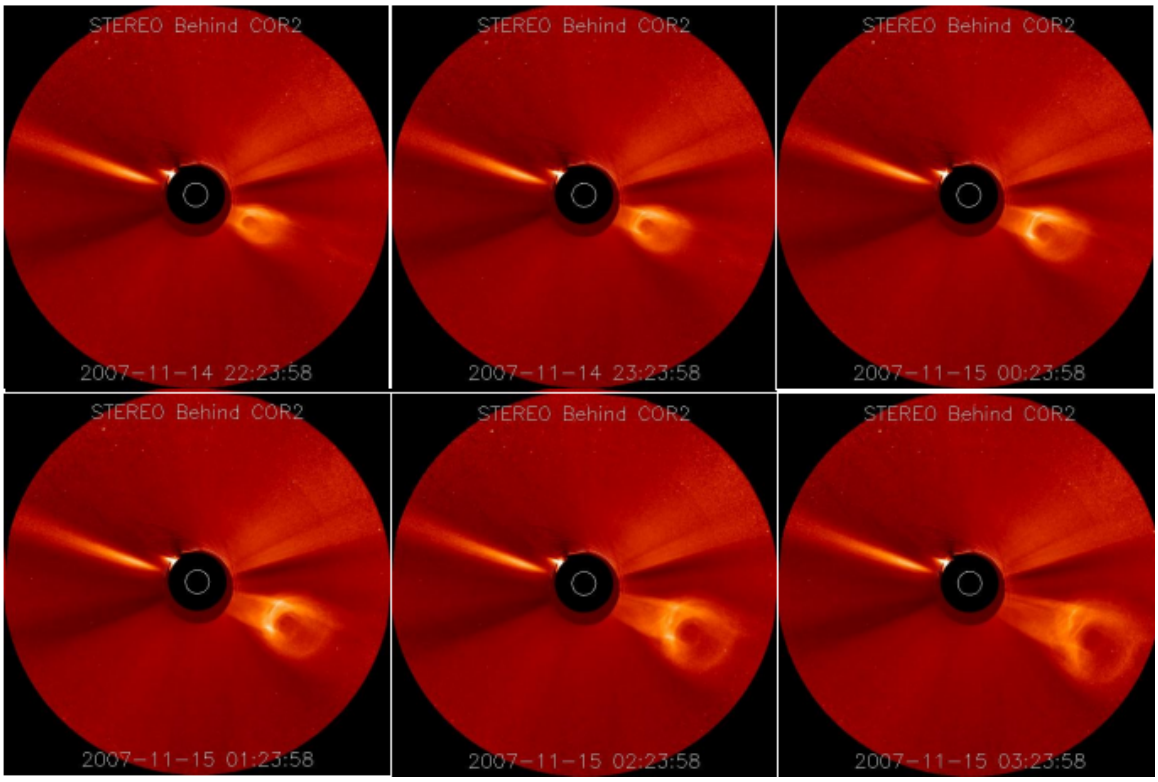


FIG. 4.1. Coronagraph Imagery, at six timesteps. One hour between each time step. The CME is propagating outward from the sun, which is represented by the white circle in the center of the image. The black circle is the occulting disc, to block the bright light from the solar surface. [Savani 2010]

movement of a CME can be captured using Coronagraph imagery. Coronagraphs depict the structure of CMEs, at varying fields of view. The data used here is designed by the developers to visualize a replication of Coronagraph imagery. An example of a Coronagraph with time variance is shown in Fig. 4.1.

4.1.2 Statistics

The data created by the CCMC was converted into .raw binary, and stored as byte data. It can be read and stored in an array of size $x * y * z$. The dataset is a uniform cube of $87 * 87 * 87$ voxels, for a total of 658,503 voxels. Each voxel represents the density of the

coronal mass ejection plasma and the surrounding background solar wind plasma (which is usually much lower), at a certain [x,y,z] grid position. The spacing between voxel samples is .25 solar radii. This is about 173,925 kilometers. So, the total length of a side of the cube of data is 15,131,475 kilometers. The center of the Sun is at the center of the cube.

There are five timesteps in my dataset. They represent five snapshots over a window of four hours. Each instance of the voxel cube represents one hour of progression of the data. So, when going from $t = 1$ to $t = 2$, one hour has passed. The same when going from $t = 2$ to $t = 3$, and so on. This is the same timescale as is displayed in Fig. 4.1. The more sparse the time steps are, the more likely the simulation will be to miss important events. Using a time disparity of one hour coincides with the duration of potential coronal mass ejection events. A higher density timescale would have been even better, but the simulation software is computationally expensive, and generating data with less than one hour between each step was not practically feasible.

4.2 Function Fitting

4.2.1 Quality Evaluation using R^2

Initially, being able to view the finished, functioning render of the PVM method, and compare it to the same visualization using the interpolation method should give a sense of how well PVM works. However, an informal evaluation only goes so far. Instead, empirical measurements give an unbiased estimate of how well the PVM approach represented the input data. Since PVM is, at its core, an approximation method to increase performance, we need to be able to tell how well the approximation worked.

For regression mapping, there exists a standard for statistical measurement of how well a predicted model fits actual data: R^2. R^2 ('R-squared') is also known as the coefficient of determination [Skovgaard 1998]. Its purpose is to check how well a regression line fit

a set of data. It does so by comparing the levels of variation in a prediction model against one another. So the variation in the dependent variable is checked against the predictabe variance from the independent variable [Skovgaard 1998].

R^2 ranges from 0 - 1 (or, equivalently, 0% - 100%). An R^2 value of zero indicates that the functional representation shares no correlation with the data at all. A value of one indicates that the function matches the data exactly. The only possible way to have an R^2 value of 1 is if the function line passes through all points of input data. In the test dataset, that would mean that the cubic polynomial found by the regression includes all five voxel values, at $t = [t_1..t_5]$. Any value of R^2 between zero and one is a measurement of how well the variables relate, with a greater value meaning greater correlation.

To calculate R^2, the differences between the expected values, and the actual values from the input must be compared. This difference is known as the residuals. Additionally, the total variance in the dependent variable must be found. When these two values are computed, they can be divided to get a percentage value. The percent represents what percentage of total variation in the dependent variable is described by the function. In other words, how well the function fit. So, we have

$$R^2 = \frac{SSR}{SST}, \tag{4.1}$$

where SSR represents the sum of squared residuals, and SST represents the total sum of squares. SSR can be calculated by summing the squared difference of all of the function output values at the exact times of each i timestep, $\hat{y}_i$, and the average output value $\bar{y}_i$. SST can be calculated by summing the squared difference of the actual data values y_i and the average output value. Thus, we have

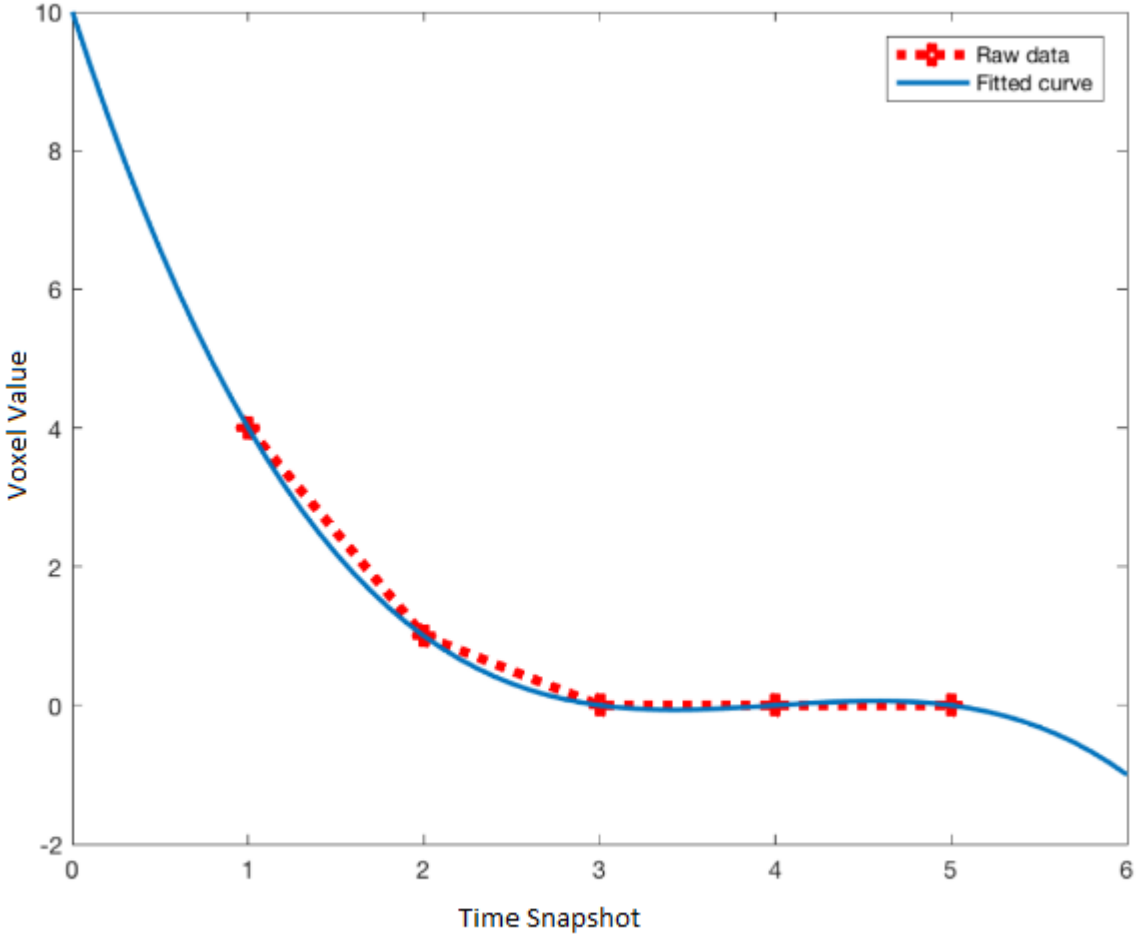
$$R^2 = \frac{SSR}{SST} = \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2} \tag{4.2}$$

where

$$\hat{y}_i = f(t_i) \quad \text{and} \quad \bar{y}_i = \frac{\sum_{i=1}^{n} y_i}{n}. \tag{4.3}$$

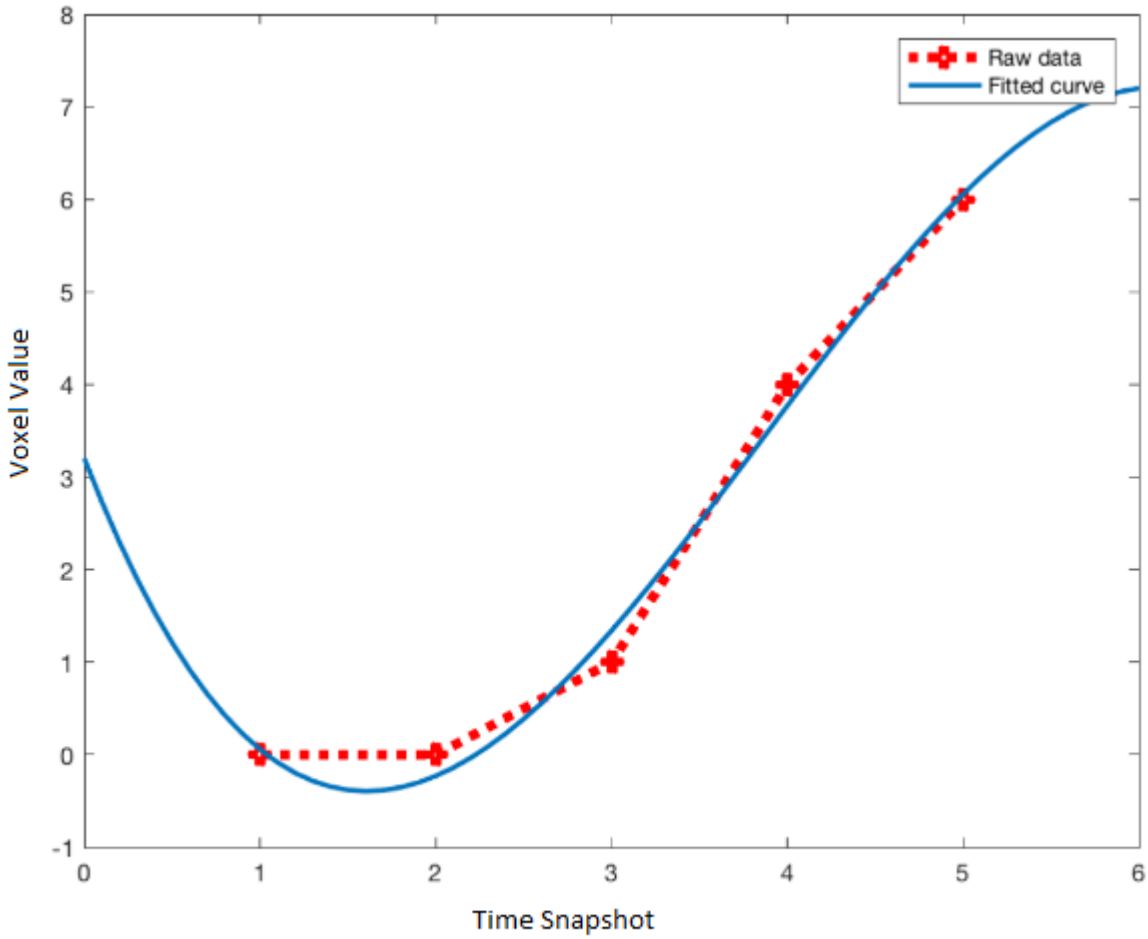
This R^2 value can be calculated and stored for every voxel, during the precomputation of the data. Then, the array of R^2 values can be analyzed to see how well the PVM calculations fit the data.

There are differing opinions on what percentage value for R^2 is ”acceptable” or ”good” [Frost 2017]. For the purpose of this thesis, I considered anything above a 90% fit rate to be a strong correlation. The following are a few example graphs of voxels taken from my implementation, that were all above an R^2 value of 90%.



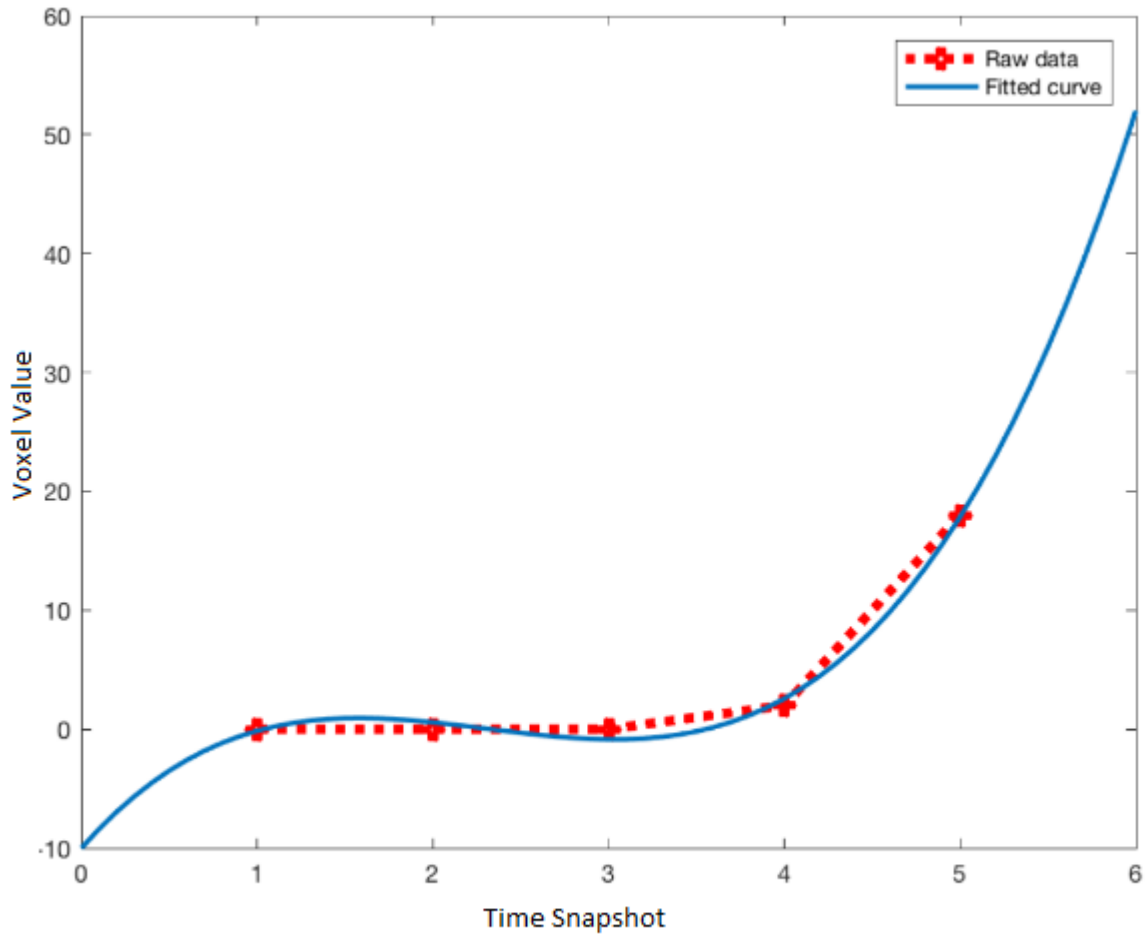
$$y_t = -0.2t^3 + 2t^2 - 7.9t + 10,\ R^2 = 100\%$$

FIG. 4.2. A voxel in the dataset with a perfect fit. $R^2 = 100\%$



$$y_t = -0.2t^3 + 1.9t^2 - 4.9t + 3.2,\ R^2 = 99.21\%$$

FIG. 4.3. A voxel in the dataset with a very close fit. $R^2 = 99.21\%$



$$y_t = 1.2t^3 - 8.1t^2 + 16.8t - 10,\ R^2 = 99.42\%$$

FIG. 4.4. A voxel in the dataset with a very close fit. $R^2 = 99.42\%$

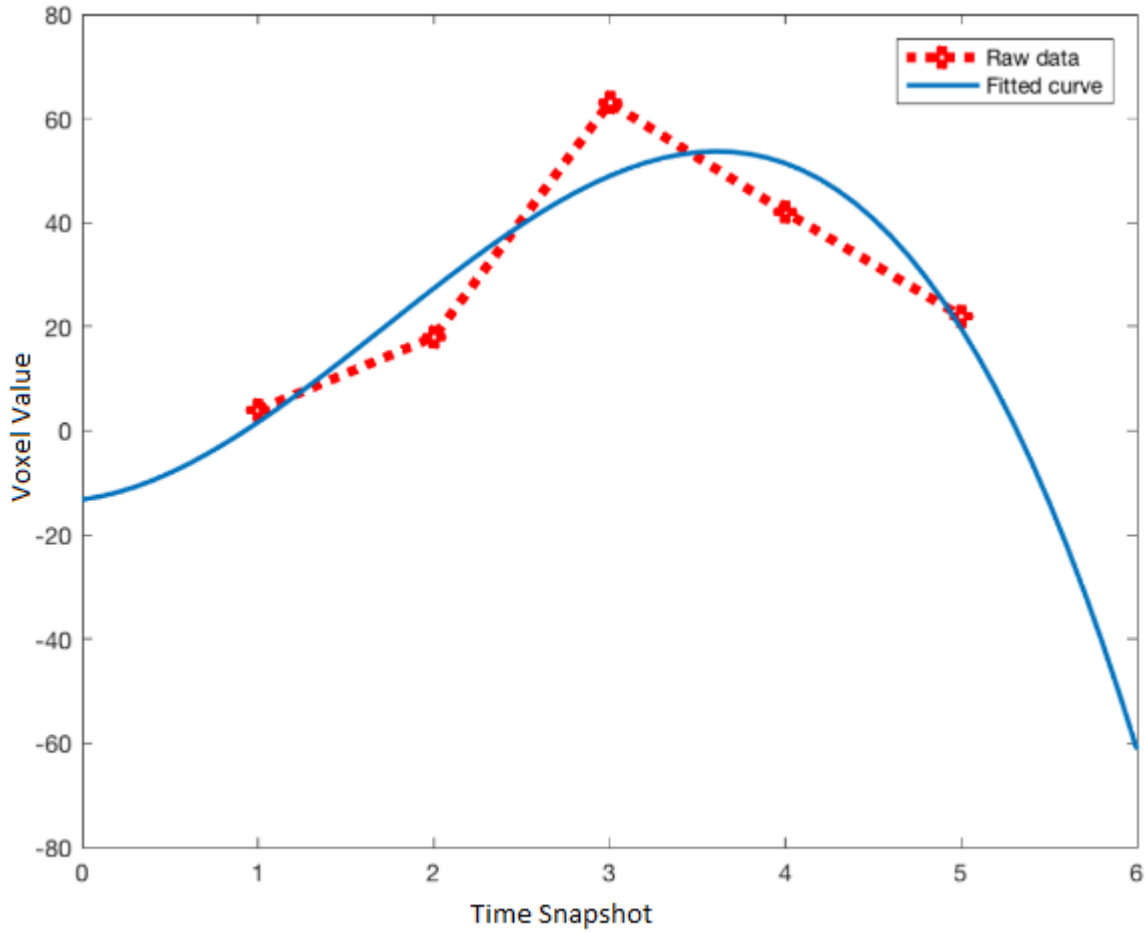
The functional representations in Fig. 4.2 – Fig. 4.4 are very close to the 5 voxel input values. In Fig. 4.2, a perfect fit of R^2 = 100% is achieved, as the fitted cubic passes through all five data points exactly. While it might seem that these graphs are simply showing a few of the best fits that the algorithm displayed, this level of accuracy is actually the standard for my dataset. Of the 658,503 voxels in the set, 632,835 created polynomial functions with an R^2 above 90%. This is 96.1% of the data. These results are shown in Table 4.1,

and indicate a strong mapping of the data to the PVM cube.

Table 4.1. Distribution of R^2

Range of R^2	Count	Percentage
$\geq 90\%$	632835	96.1%
$\geq 80\%$	9389	1.4%
$\geq 70\%$	10202	1.5%
$\geq 60\%$	1168	0.2%
$< 60\%$	4909	0.7%

Even still, the mapping was not perfect. There are some voxels in the set that displayed an R^2 value less than 90%. 9,389 voxels (1.4% of the data) were between $R^2 = 80\%$ and $R^2 = 90\%$. An R^2 value in this range still constitutes a decent fit of the data. An example of a voxel with a fit in this range is shown in Fig. 4.5.



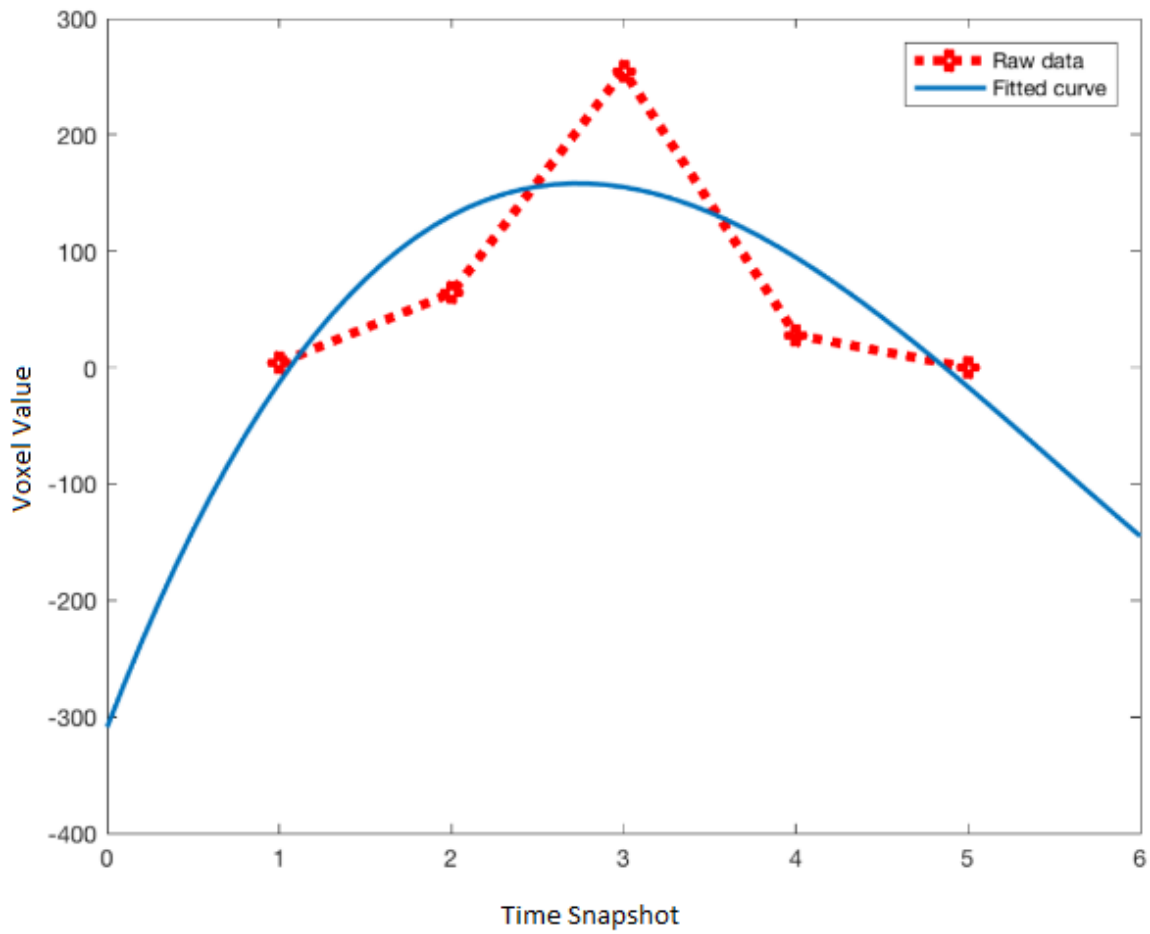
$$y_t = -2.5t^3 + 12.9t^2 + 4.4t - 13.2,\ R^2 = 81.85\%$$

FIG. 4.5. A voxel in the dataset with a decent fit. $R^2 = 81.85\%$

Unfortunately, some of the voxels contained an R^2 that was quite low. In this context, low is defined as anything $\leq 60\%$. These functions do not appear to map to the data very well, and are typically associated with an abrupt spike in the density of a voxel. In this sense, since the 'spike' has been smoothed out, some information may have been lost in the process of converting from raw data to PVM.

The total number of voxels containing data with an R^2 of 60% or below was 4,909.

This is 0.7% of the data, which is a very small portion. One of these voxels is shown in Fig. 4.6.



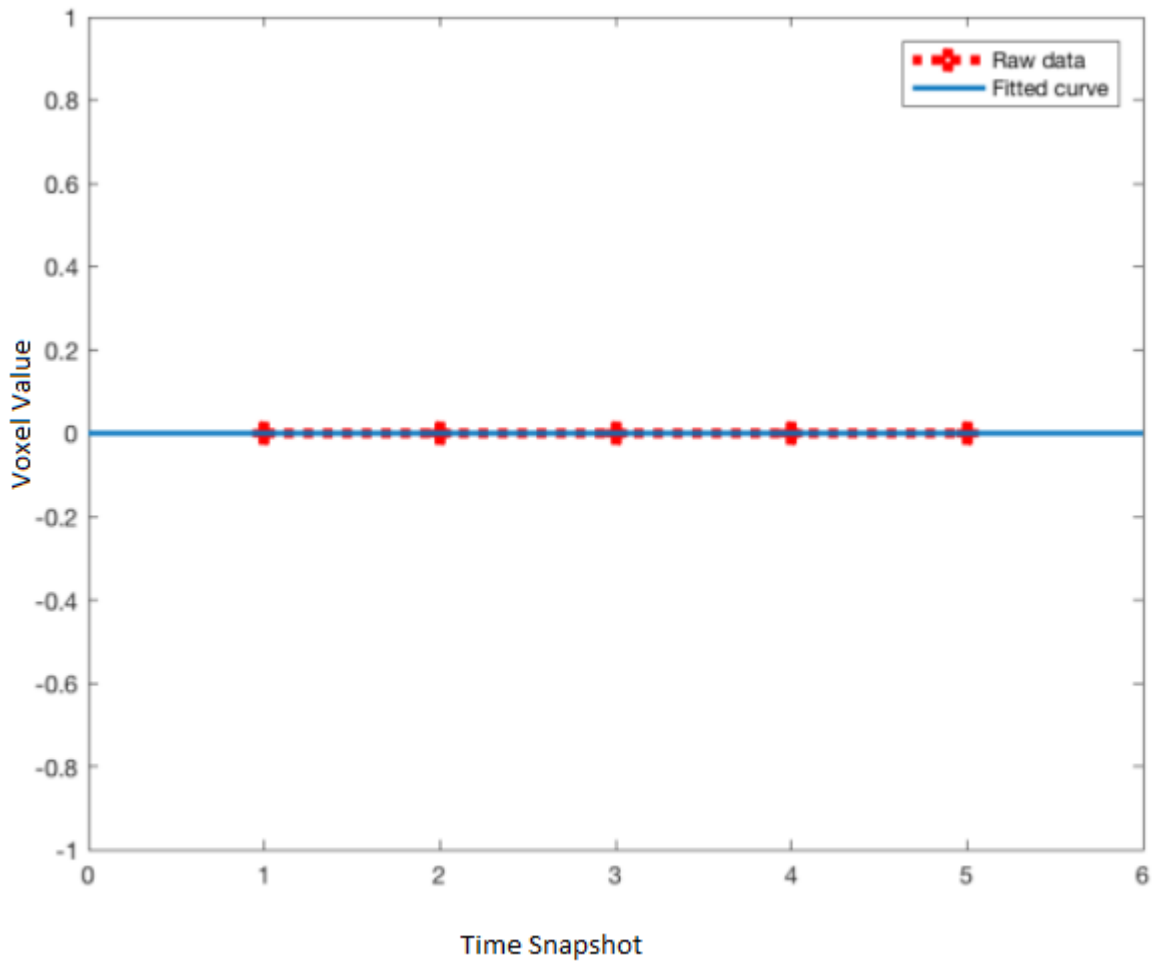
$y_t = 5.7t^3 - 93.4t^2 + 383.9t - 308.8,\ R^2 = 57.11\%$

FIG. 4.6. A voxel in the dataset with a bad fit. $R^2 = 57.11\%$

4.2.2 Empty Space

There is one more factor that must be considered, when judging how well the function fitting worked. While a percentage of 96.1% of the data being at an R^2 of 90% or above seems high, this number is inflated. Much of the dataset is empty space. Many of the

voxels contain just zeros, because no part of the CME is included in that voxel at any point during the four hours. These magnify the total R^2 number, because they all contain an R^2 of 100%, as seen in Fig. 4.7. Given five input data points, all of which contain a value of zero, the regression calculation will run successfully. However, the result is trivial. If there are 5 zeros, the cubic polynomial that fits the data will always be $0t^3 + 0t^2 + 0t + 0$. A straight line through $f(t) = 0$.



$y_t = 0^3 + 0t^2 + 0t + 0,\ R^2 = 100\%$

FIG. 4.7. An empty voxel. No interesting data.

In total, of the 658,503 voxels, 568,277 of them were all zeros. This means that 86.3% of the dataset was empty space. To get a more accurate idea of how well the function mapping fit the data, these empty voxels need to be excluded. So, we need to focus purely on the 90,226 voxels (13.7% of the total data) that contain data that changed. To enforce this restriction, the number of all-zero voxels can be subtracted from the number of voxels with an R^2 of 90% or higher, as they were all included in that figure. This leaves the dataset with 64,558 voxels that contained interesting time-varying data, with an $R^2 \geq 90\%$. Table 4.2 gives a breakdown of zero vs. interesting voxel counts.

Table 4.2. Dataset Summary

	Count	Percentage
All zeros	568277	86.3%
Nonzeros	90226	13.7%

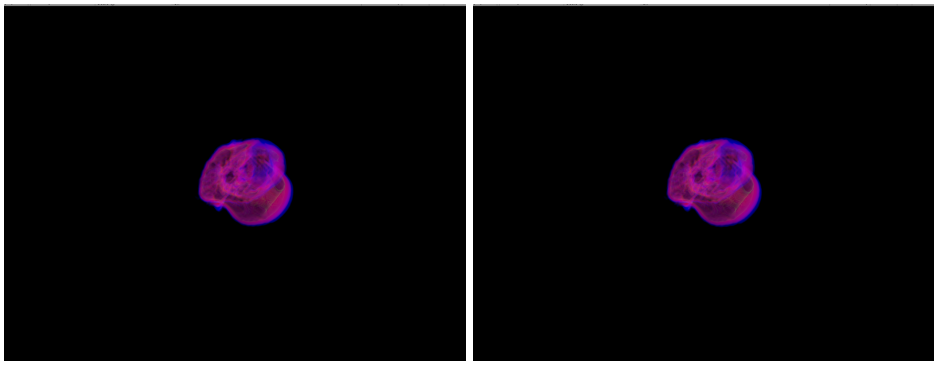
Once the zero voxels have been excluded from the 'good' pool, we can also exclude them from the total, for percentage calculations. Of the 90,226 voxels of interest, 64,558 of them were $R^2 \geq 90\%$. This is still 71.6% of the data. If the definition of 'good' is extended to include everything with an $R^2 \geq 80\%$, this figure jumps to 82.0% of the data. The amount of voxels with a 'bad' R^2 of below 60% remains at 4,909. This is 5.4% of the interesting voxels, which is still quite low. The results excluding all-zero voxels are shown in Table 4.3.

Table 4.3. Distribution of R^2, excluding zeros

Range of R^2	Count	Percentage
$\geq 90\%$	64558	71.6%
$\geq 80\%$	9389	10.4%
$\geq 70\%$	10202	11.3%
$\geq 60\%$	1168	1.3%
$< 60\%$	4909	5.4%

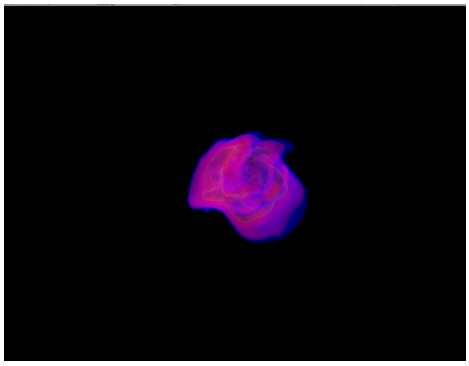
4.2.3 Screenshots

The following is a side-by-side comparison of screenshots, taken at the same temporal data progression, of the interpolation method vs the PVM method. In the left column, the interpolated method, at different levels of interpolation. In the right column, PVM method at the corresponding t value.

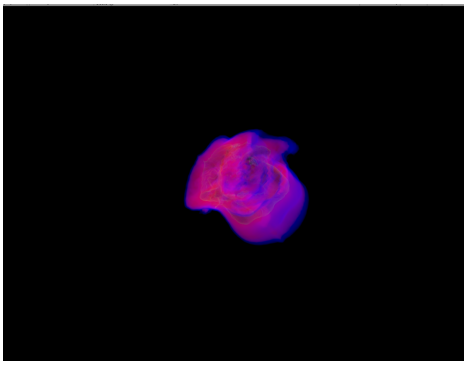


(a) test

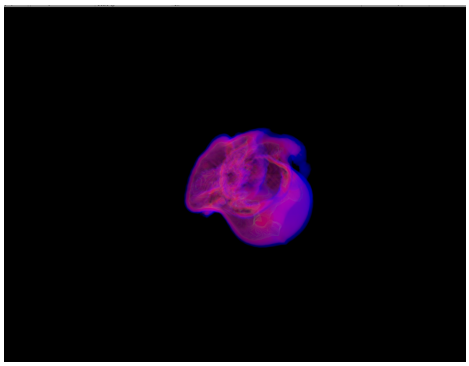
(b)



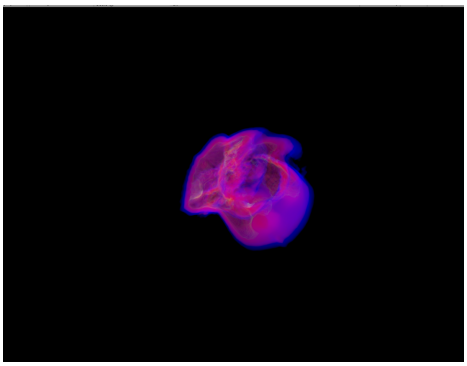
(c)



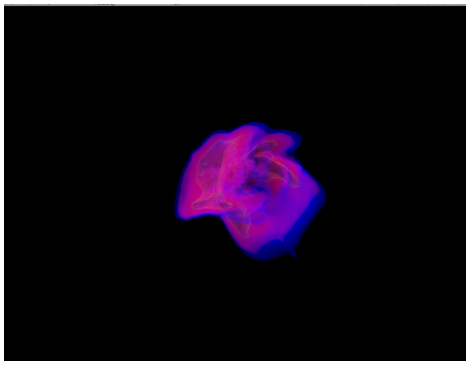
(d)



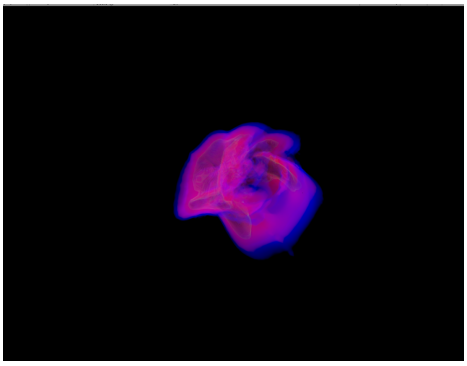
(e)



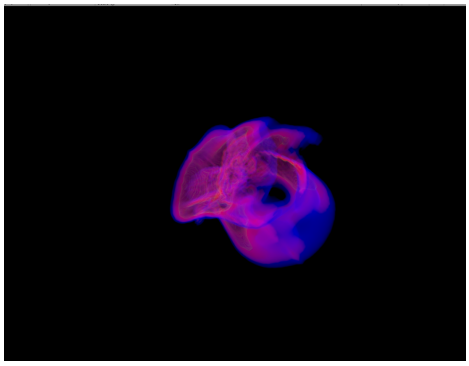
(f)



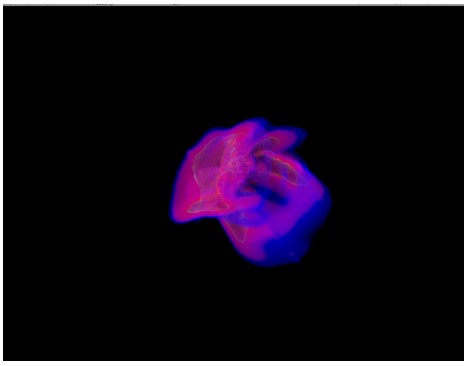
(g)



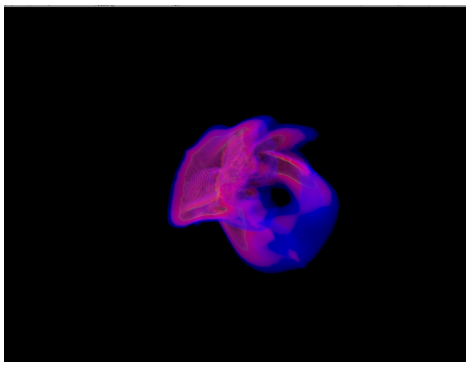
(h)



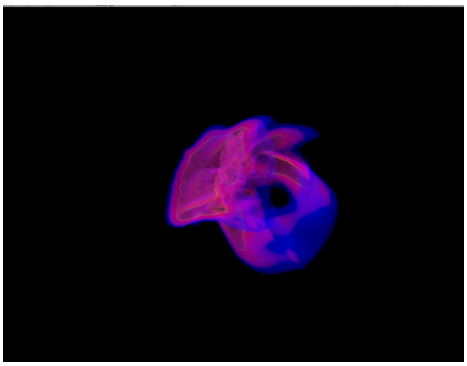
(i)



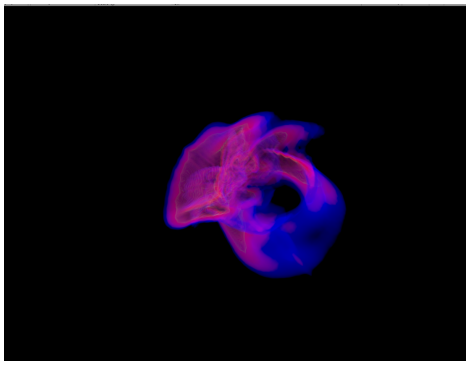
(j)



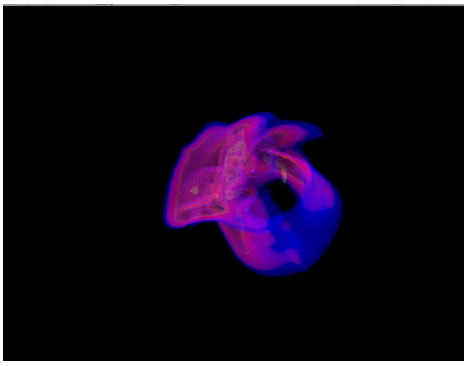
(k)



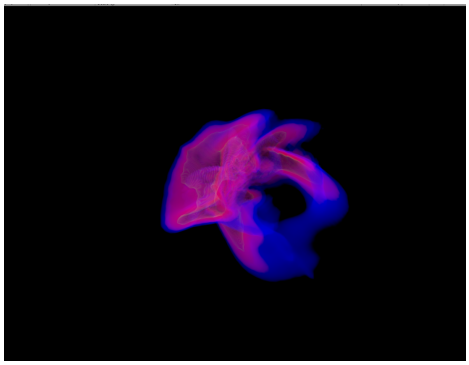
(l)



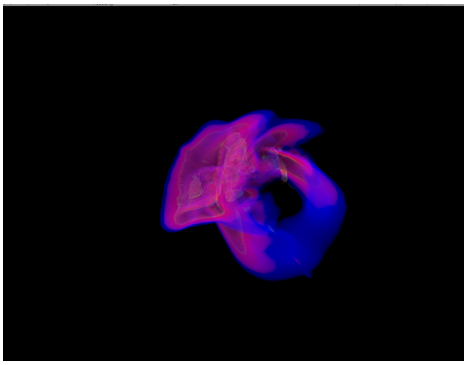
(m)



(n)



(o)



(p)

4.2.4 Error Visualization

Similar to the side-by-side screenshots showcasing differences between the two methods, it was possible to visualize the R^2 measurements per voxel. This allows a 3D representation of the concentration of good and bad functional mappings. A low R^2 value translates to a high density for the voxel, and a high R^2 value maps to a low density. When this data is written to a volume and passed as a 3D texture to the same ray casting renderer, it can be displayed in the same dimensions as the original data. This is shown in Fig. 4.8.

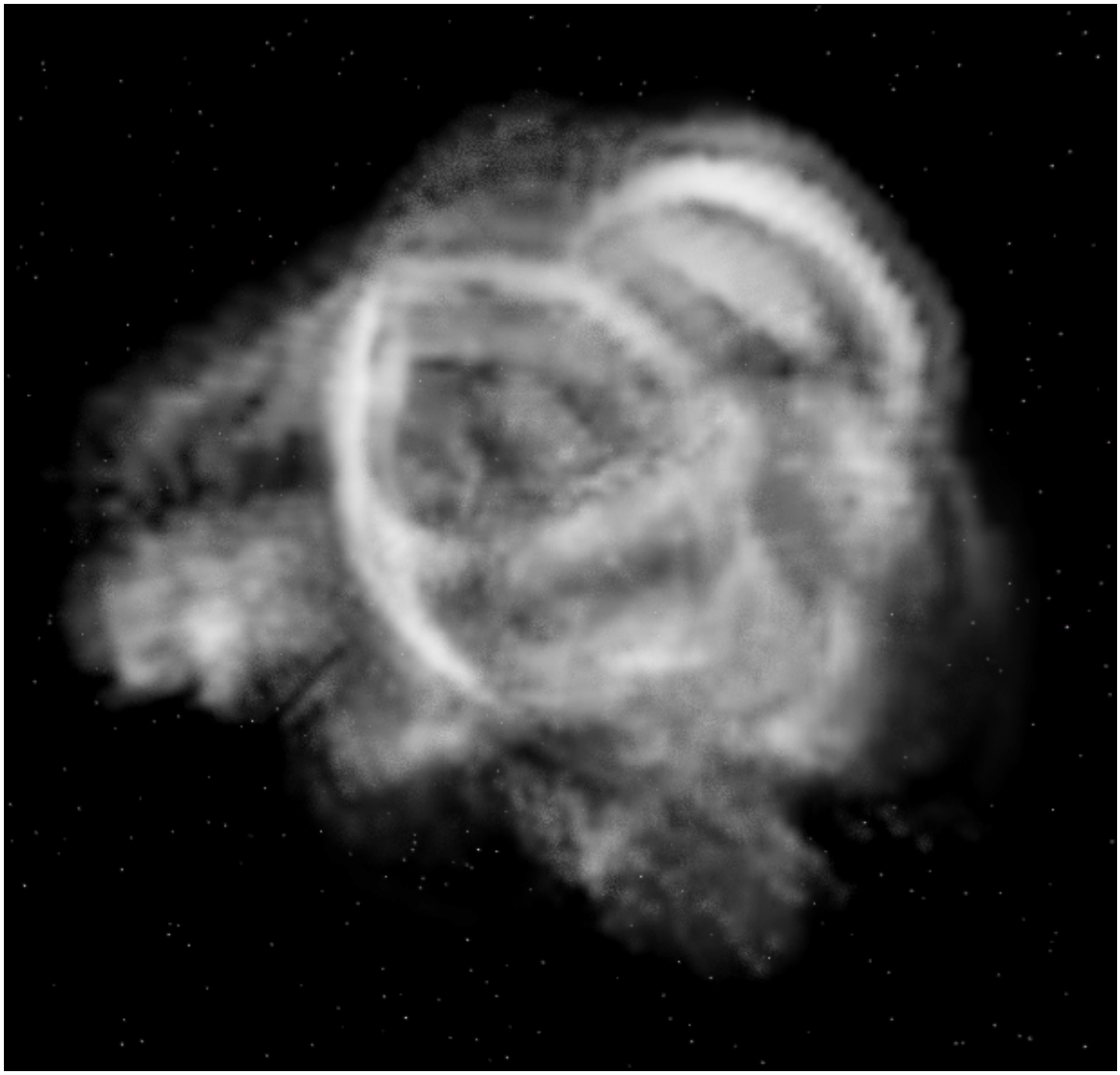


FIG. 4.8. R^2 error visualization. High opacity indicates a poorer functional mapping.

As expected, in the outer shell sections of the data, the error volume is empty. This is due to the perfect mapping from having all zero voxel steps. The image is more interesting towards the center of the volume. Arc structures can be seen with high density values, indication a poorer function fit. This is likely due to sharp transitions in the volume display, where one timestep spiked much higher or lower than the two adjacent steps. An abrupt

change like that is expected to skew the funtion fit, and results in a much more visible error display.

4.3 System Specifications

The test computer was provided by NASA Goddard Space Flight Center's STEAM Innovation Lab. It is the computer used for Virtual Reality demos in the lab. It is also the computer where much of the development work took place. The specifications of the computer are given in Table 4.4

Table 4.4. Test System Specification

Component	Hardware
CPU	Intel Core i7 6700K (Skylake) 14nm LGA 1151 (Socket H4) 4.0GHz, 4 cores, hyperthreaded 8 threads.
Memory	16GB DDR4 @ 2133 MHz
GPU	NVIDIA GeForce GTX 1080 (Pascal) 16nm 1709 MHz base, 1848 MHz boost 8GB GDDR5 @ 1251 MHz
Disk	Western Digital Blue HDD 7200 RPM - 1 TB
Display	BenQ ZOWIE XL2730 - 1920x1080, 144hz HTC VIVE HMD - 2160x1200 (1080x1200 per eye), 90hz

4.4 Performance

4.4.1 Frame Timings

Frame timings are very important when evaluating the performance of GPU rendering [Dinh et al. 1999]. The frame time displays how much time it took to render and display a single image frame to the user. These frames are displayed at a rate, called Frames Per Second (FPS). The longer it takes to render a single frame, the lower the FPS must be, as a frame cannot be displayed until it has been fully rendered.

Tos support Virtual Reality using the HTC Vive, we must maintain a framerate of 90 FPS. The Vive has a refresh rate of 90 Hz, meaning it can display a maximum of 90 images in one second. This high refresh rate is important for virtual reality experiences, to keep the user immersed in the virtual world. When images are displayed slower, it can cause anything from mild annoyance, to headaches and nausea [Waters 2015]. To maintain 90 FPS, a frame time of no more than $1/90$ seconds is needed. This is 11.1 milliseconds. So, as long as the render times are below 11.1 milliseconds, the viewer will see no conceivable difference.

The difference in frame times, in this test scenario, can come only from differences in the GPU computation. The camera was locked at a fixed orientation, so that no camera movement could affect the rendering time. The same cube was inserted at the same 3D world position, with the same orientation. The only difference is the volume interpolation method being applied.

One of the cubes uses the interpolated method, the other uses the PVM approach. The main difference, code wise, in these two renderers is how the lookups into the volume data are handled. In the interpolation code, two volumes must be sampled into to acquire data at a frame: the previous timestep, and the next timestep. Both samples are taken, and then an interpolation call is executed. In the PVM version, only one lookup is needed, into the

PVM cube of coefficients. Then, instead of a linear interpolation, a cubic polynomial is evaluated.

The final aspect of these timings is the inclusion of both 'static' and 'rotating' measurements. The static measurements are taken from a fixed camera position, to a fixed volume orientation. The rotating measurements are taken with the volume slowly rotating in place. This was to get a better overall feel for the variability in frame times, as looking at a volume from a different angle changes all of the view vectors, and can change how many steps into the volume must be taken. The timings are recorded at varying levels of max sampling rates. The value listed is the maximum steps taken, and the steps are adjusted to fit within the volume, corner to corner. So, more samples also means a smaller step size.

The results are shown in Tables 4.5 – 4.9

Table 4.5. GPU Frame Time – milliseconds (50 samples)

Category	Min	Max	Average
Lerp static	3.566	5.102	3.813
PVM static	2.509	3.512	2.588
Lerp rotating	3.571	5.562	3.991
PVM rotating	2.522	3.657	2.624

Table 4.6. GPU Frame Times – milliseconds (100 samples)

Category	Min	Max	Average
Lerp static	8.421	9.183	8.891
PVM static	4.611	5.061	4.694
Lerp rotating	8.569	10.254	9.153
PVM rotating	4.604	5.901	5.059

Table 4.7. GPU Frame Times – milliseconds (300 samples)

Category	Min	Max	Average
Lerp static	15.174	20.872	18.143
PVM static	14.827	15.476	15.017
Lerp rotating	14.942	23.129	19.639
PVM rotating	14.822	17.769	15.580

Table 4.8. GPU Frame Times – milliseconds (500 samples)

Category	Min	Max	Average
Lerp static	25.461	26.832	25.701
PVM static	25.336	26.718	25.623
Lerp rotating	25.573	30.694	27.672
PVM rotating	25.536	231.258	27.395

Table 4.9. GPU Frame Times – milliseconds (1000 samples)

Category	Min	Max	Average
Lerp static	50.142	52.490	50.832
PVM static	50.458	52.160	50.851
Lerp rotating	50.279	58.848	53.627
PVM rotating	49.724	58.014	52.738

While it was not the intent of the PVM approach, and did not contribute to the reasoning for development, the frame timings were faster for PVM than interpolation. The most noticeable improvement was in the 100 sample range, which is closest to how these volumetric ray marchers will be used in real-world applications, on modern day hardware. This level of sampling allows for good looking results, while not monopolizing the GPU entirely. The increase in average frame times for the rotating interpolated approach over the PVM approach is quite large, increasing from 5.059 ms to 9.153 ms when rendered via interpolation. The major difference here is the requirement of the interpolated method to do two texture lookups per step, rather than just one. So, for each of the 100 samples, there were twice as many texture references, 200 instead of 100. While it is impossible to say for sure, with the analysis software I have, this performance difference was likely a result of GPU memory cache misses. When consistently indexing into two different volumes instead of one, the cache hit rate will be lower. While the frame times were below the 11.1 ms threshold for VR, this was a very controlled set up. In a real application, the GPU would be taxed by other tasks as well, meaning the extra frame time within the 11.1 limit could be put to use doing other things.

4.4.2 Execution Time

The primary advantage of PVM over interpolation is the ability to span a larger number of frames without loading. The frame rendering performance was expected to be relatively similar, as most of the math carried out in the two approaches is unchanged. Other than the interpolation code, the ray marching algorithm is the same in both implementations. So, instead, the gain in performance is the ability to execute the volume rendering code without having to wait for new data to be loaded and passed to the graphics card.

To evaluate this the benefit, performance was measured via execution time. The same testing environment as the frame timing measurements was used. However, this time a sampling rate of 150 steps was used across all tests. The variation came from how quickly the renderer was to cycle through the five timesteps. Disparity between time snapshots was set to be a predefined number of frames for every test case.

First, the 'start-up' time for the testing environment was recorded. This elapsed time measures the total time to get everything up and running, without actually rendering any frames of the volume. This accomplishes two things: it creates a baseline for the other measurements to be compared, and it gives a time that can be subtracted from the other tests to get a time that only represents the actual volume handling. After measuring the startup time, the execution of a variety of interstep frame counts was tested, using 20 full data cycles. The execution time was the time that the application took to render a full transition of timesteps 1 through 5, twenty times.

The results of these tests are shown in Tables 4.10 and 4.11.

Table 4.10. Total Execution Time for different substep frame counts. (Seconds)

Category	Startup	1 fr.	5 fr.	10 fr.	30 fr.	90 fr.	270 fr.
Lerp	5.656	9.110	13.454	19.888	33.482	86.942	246.964
PVM	4.852	6.436	9.526	14.049	32.239	85.742	246.417

Table 4.11. Total Execution Time, accounted for Startup Time (Seconds)

Category	1 frame	5 fr.	10 fr.	30 fr.	90 fr.	270 fr.
Lerp	3.454	7.798	14.232	27.826	81.286	241.308
PVM	1.584	4.674	9.197	27.387	80.890	241.565
Difference	+118.1%	+66.8%	+54.7%	+1.6%	+4.9%	-1.1%

The PVM approach offers a significant increase in performance over standard interpolation. This is particularly true when the timesteps for the data to be rendered are temporally dense. When steps must be cycled through quickly, the interpolated method simply does not work. The rendering becomes bottlenecked, having to constantly wait for the memory loading to complete. To the end user, this creates a slow, choppy visualization.

The faster transition rates received a huge performance increase when using PVM. At the lowest possible intermittent frame count, where each piece of input data only exists for a single frame, there was an improvement of over 118% in execution time, if startup is discounted. If each snapshot of volume data only exists for one frame, then every single frame a new one must be loaded. Thus, during every frame, the renderer must wait for the data to be loaded and passed to the GPU before doing any sampling. The memory load bottlenecks the rendering immensely.

That is the best case scenario for PVM. If a dataset has a new timestep every $1/90$ seconds, the interpolation approach cannot be applied, but PVM can. For the next tested counts, which are still high temporal density, performance was also increased. 5 and 10 frame transitions were improved substantially, with performance boosts of 66.8% and 54.7% respectively. This means that the PVM method allows the data to be cycled through at a much higher rate than standard interpolation. When observing the rendering, using the HTC Vive, the interpolated method appeared very sluggish, and consistently stalled, while the PVM method ran at a smooth 90 FPS.

It is worth repeating that the input dataset was 87*87*87 voxels. If a larger dataset was used, such as 256*256*256, the memory bottleneck would be more severe. 87^3 is 658,503, while 256^3 is 16,777,216. Using a single float per voxel, this is a difference of 2.512 megabytes vs 64 megabytes. The 256^3 dataset would require loading 64 megabytes of memory (25 times as much), in theoretically the same amount of time. This is not possible. Slowdown is going to happen if that much data must be loaded during execution, in such a small window of time.

When allowing 30 frames for the loading of the next volume timestep, the execution time was almost the same between the interpolation and PVM approaches. This can be interpreted as the memory bottleneck not existing, when enough time is allocated to loading the next volume. There was almost no performance change in any frame amount above 30, because the extra time is irrelevant. Once the data is loaded, it simply must be swapped out to the interpolation function. This can easily be done in a tiny fraction of a single frame, by the renderer. Any extra time to load is unneeded. No memory is being transferred during that time, hence the very small variability after the 30 frame mark.

4.4.3 Memory

The range of potential values for the computed coefficients is quite large. See Table 4.12. For this reason, it is required to use at least a 4-byte single-precision floating point value to store the values. Using a smaller bit representation to save memory (such as a 2-byte float) limits the coefficient range too much. Thus, the memory overhead required to store the results of the computation is $numBytes * numCoeffs * numVoxels$, or $(4 * 4 * x * y * z)$. For the test dataset, this comes out to 10,536,048 bytes, or 10.0 megabytes.

Table 4.12. Range of Coefficient Results

	Max	Min	Average
A	63.74315	-63.742	0.014403
B	573.0749	-563.612	-0.24257
C	1519.94	-1598.51	1.52356
D	1352.069	-1044.87	0.032925
Overall	1519.94	-1598.51	0.332078

Given that the 3D textures use color values that are actually 4-byte floats, this was also the size of the constructed textures that were passed to the GPU. While evaluating execution time, the RAM and VRAM usage were also monitored. Using the benchmarking tools at my disposal, I was able to track the maximum usage of CPU and GPU memory over the running time of the application, which was the same 20 cycles of data. The results are listed in Table 4.13.

Table 4.13. Maximum Memory Utilization (Megabytes)

Category	RAM usage	VRAM usage
Lerp	90.3 MB	33.1 MB
PVM	80.3 MB	23.1 MB

The memory usage during execution did not fluctuate much from the maximum values listed here. RAM usage refers to the CPU memory, which is only used to load in the volumes from disk, and pass them to the GPU. The interpolation method was 10 MB greater than the PVM method, because of the initial data handling on program launch. At the start, two volumes must be completely loaded and passed to the GPU — step 1 and step 2. Afterwards, the CPU memory can be cleared and the third volume can begin to be loaded. Since the PVM approach only requires the PVM cube to be loaded and passed, the additional 10 MB is not needed. This assumes that the PVM data has already been precomputed. The memory required for the precomputation may differ, but this does not factor into the volume rendering.

For VRAM, the interpolation method also took 10 MB more memory than the PVM approach. This is an increase of 43.3% in total memory usage. This increase stems from the requirement of having two volumes always loaded into VRAM, for interpolation. In this case, the PVM approach requires a single volume, and the interpolation approach requires two, so the difference was the size of a single volume of data.

All of these values are relatively small when compared to the total VRAM capacity on the GPU used in testing, which is 8 gigabytes. A larger dataset would certainly require more memory. Though, the main benefit of the PVM method is the reduction in runtime load operations. The amount of memory that must be passed to the GPU during program execution is drastically reduced. Unfortunately this was not tracked with a benchmarking

tool, but simple calculation can give a rough estimate of the difference in total data loaded from disk for this test setting. For the PVM approach, no data was loaded after program launch, so this value will remain at 23.1 MB of VRAM. For the interpolation method, each cycle requires loading 5 volumes with a size of 10 MB, so the total size for a single cycle is 5 * 10 = 50 MB. Since the data was cycled 20 times, the cumulative amount of data passed to the GPU for rendering is 50 * 20 = 1,000 MB. This can be added to the 13.1 MB of VRAM used for other things within the test environment, to get 1,013.1 MB, or roughly 1 GB. That is an increase in total memory of 4,787%. These values are shown in Table 4.14.

Table 4.14. Total Memory passed to GPU (Megabytes)

Category	Total memory
Lerp	1013.1 MB
PVM	23.1 MB

While 4,787% seems like a massive figure, it is a measurement of total memory passed to the GPU. Unless the act of loading the data bottlenecks the execution of the application, the extra memory usage is acceptable. As long as execution continues smoothly, and 90 FPS is maintained, there is no conceivable difference. Still, being able to evaluate the amount of memory used is useful for analysis. The increase in concurrent memory usage is more substantial than the increase in total memory passed, because when dealing with large datasets, it can become restrictive. There was a 43.3% increase in memory taken by the application for the interpolation method, which could cause a large dataset to not fit into VRAM when using the interpolation method, even if it would fit when using PVM. Additionally, as the dataset grows in size, it will be proportionally greater than the VRAM taken by other things on the GPU, so it would likely approach 100% increase in memory

asymptotically, due to requiring two volumes instead of one.

4.4.4 GPU Utilization

Yet another measurement of performance when it comes to GPU-based rendering is the overall utilization of the graphics card itself. This is represented by a percentage load, which is an aggregate rating to tell how hard the GPU is working at a particular moment in time. Typically, when the GPU load gets into very high numbers, frame timings begin to suffer, as the GPU cannot keep up with the computational requirements of the application.

Using the same testing environment that was used for frame timings, I recorded the GPU utilization of the shaders, with varying sample rates. Higher step frequencies will push the GPU harder, as it has to do more computation within each frame. Tables 4.15 and 4.16 display the same testing conditions for the interpolation method, and then the PVM method.

Table 4.15. Lerp GPU Utilization (Load %)

Number of Samples	Average	Max
50	37	42
100	61	69
300	88	93
500	97	98
1000	99	99

Table 4.16. PVM GPU Utilization (Load %)

Number of Samples	Average	Max
50	32	36
100	52	58
300	72	82
500	97	98
1000	99	99

The values for GPU usage are relatively close, but the PVM implementation consistently edges out the interpolated one. Again, this is likely due to the interpolation approach requiring two texture lookups per march step. In the lower sampling rates, the GPU runs very comfortably in both setups. Around the 300 steps mark, both implementations start to suffer, with the interpolated method causing an 88% load on the GPU, and the PVM causing 72%. Above this amount, both implementations tax the GPU in its entirety, and the frametimes drop significantly.

Chapter 5

CONCLUSION

5.1 Large Time-varying Volumes

The reduction of execution time loads offers the most benefit when dealing with large data sizes. The more voxels in a dataset, the more raw bytes must be loaded from disk, converted to a texture, and passed to the GPU. Without the PVM approach this must be done once for every timestep, during program execution. Using the example listed in the previous section, of a dataset with 256^3 voxels, this amounts to 64 megabytes of memory for every step, if each voxel is a 4-byte float.

When PVM is used, however, the load overhead is substantially lowered. Once the volume has been fit to a PVM volume, only the PVM volume must be passed. This compresses multiple timesteps into a single volume of the same dimensions. So, when dealing with large datasets, that are to be rendered using multiple timesteps, the PVM approach is preferred. As long as the computed polynomial functions fit the data adequately, much of the load overhead can be eliminated.

This ability to handle larger volumes, because of the reduction of runtime loads, lends itself well to large scientific datasets. The CME data used for testing is just one of many time-varying datasets available for visualization at NASA Goddard. Other applicable data includes simulations from the CCMC, and the Magnetospheric Multiscale Mission (MMS)

team [NASA 2018b].

5.2 Future Work: Functional Data Mapping

Much of the available data has a very high number of timesteps, which will cause a problem with the implementation used for testing in this thesis.The PVM approach relies on fitting a single 4-coefficient cubic function to the data. If a cubic line does not fit the data well, the margin for error will be large. Using a set of data with many steps, a cubic will not be able to effectively visualize the shape of the data. As an example, what would happen using this approach with a set of voxels that resembles a sine wave, with values fluctuating up and down over a long stretch of time? A cubic function would not be able to effectively map all of the data in this case, and the PVM visualization would look very different (and incorrect) from the interpolated one.

5.2.1 Piecewise Alteration

There are many variations that can be made to the function fitting section of PVM, without changing the idealogy. The concept of substituting a fitted function for raw data values carries the same benefits, even if some aspects of the PVM approach are altered. No changes should be needed for data that maps well to a single cubic polynomial, as is shown in the results for this thesis. However, for datasets that map less accurately, or have a long time window, other approaches can be considered. One is to use multiple cubic polynomials, in a piecewise fashion, instead of one function mapped to the entirety of the data.

If this approach were to be used, the data would need to be mapped in segments, rather than all at once. A certain subset of x time snapshots could be chosen, and those would be mapped to their own set of polynomial voxels. Then, the next subsequent set of x

polynomials could be mapped to their own cubic, and so on. This process would need to be modified to combine the curves into a smooth one, to avoid disconnected 'jumps' at every break between PVM blocks. In this way, the construction of the cubes would be analagous to that of constructing a Bezier spline path, where each point contributes to the curve, along with the set of 3 points around it [Prautzsch et al. 2013]. This approach would, in theory, fix the problem of a single cubic not mapping well to the data, because the data would be divided up into small enough segments that a cubic would fit well to its own small subset.

The downside of this is that now data must be loaded to the GPU in real time. All of the PVM volumes would be pre-computed, but now there is more than one single volume that gets loaded at the start. Instead, every time a new set of polynomials is needed, a new volume must be loaded on to the GPU for sampling. Using a $x = 5$ example, the amount of loads has still been reduced by a factor of 5. This allows a considerable increase in the amount of time to load a single volume of polynomials to the GPU. Rather than loads being completely eliminated, their frequency is reduced by a large amount. As long as there is enough time to load the new PVM cube while the first one is being sampled, the performance should be better than interpolating raw voxels.

The other benefit of mapping things in a piecewise fashion, rather than a single cubic for all of the data, is that the margin for error will likely be lower. R^2 will be higher if the functions fit the data more accurately. Using a piecewise approach to fit the data, instead of mapping all of the potentially large number of timesteps to a single function, will cause the function fit lines to be closer to the actual data points [Plass and Stone 1983]. Thus, the variance in the output will be reduced, causing an increase in R^2.

5.2.2 Different Regression and/or Functions

Another change in the PVM approach that can be explored is using a different method for regression, rather than Ordinary Least Squares. There are other methods than OLS for

estimation. Bayesian linear regression is an option. It offers a potentially lower margin for error on datasets that span long sequences [Reis et al.]. Optimally, in an integrated system, multiple regression analysis methods would be tried on a dataset and evaluated for the best R^2. Then, that particular volume of cubic coefficients would be used for the GPU pass, and sampled.

Using other function models than a cubic polynomial could also be explored. This would require more modification to the rendering code, as well as the volume construction. When the volume data is sampled, the results of the sample are used to evaluate the polynomial. This routine would need to be changed to accomodate evaluation of whatever type of function was being utilized. Then, the regression mapping would need to be changed to solve for a different β set of coefficients. This could be combined with a modified regression approach to find volume GPU values completely differently than this thesis, while still maintaining the core concept of PVM.

The complication if this were to be attempted is the way that the GPU code packages the volume data. Currently, the coefficients are passed in as a single volume, with $\beta = [A, B, C, D]$ being converted to $VoxelColor = [R, G, B, A]$. Since current graphics APIs (like DirectX) only support up to 4 channels per texel, there is a limit of no more than 4 coefficients per voxel. This could be circumvented by passing multiple volumes to the GPU, and storing the coefficients across corresponding texel positions. For example, if working with polynomials, two volumes could be passed instead of one, which would double the maximum number of coefficients from 4 to 8. Then, a polynomial up to an order of 7, instead of 3, could be supported. This could offer better functional mapping, at the cost of frame timings. Using that approach, two texture samples would be required at every step instead of one, and a polynomial of order 7 would need to be evaluated rather than order 3, which puts more work on the GPU.

5.3 Future Work: Volume Rendering Addition

There is also more work that can be done on the rendering side of the PVM approach. Once the volume of precomputed data has been sent to the GPU, the way that the data gets handled is up to the shader. Currently, in the testing environment that was made for working with PVM, the ray is computed, the volume is sampled, the sample is evaluated as a polynomial with respect to time, and the result is accumulated along the ray and returned. This final result is output as an RGBA pixel color. Other than texture filtering to avoid a 'blocky' voxelized appearance, there is no additional computation applied to either the data, or the resulting pixels.

5.3.1 Volumetric Motion Blur

A conceivably applicable modification is that of *volumetric motion blur*. Motion blur is a common technique in modern graphics, that involves temporally blending frames that include fast moving subjects. The blending of the pixels creates an effect that looks more realistic to the eye. There has been some research into expanding this tactic to volumes, and blending data voxels rather than screen space pixels. One such approach incorporates a data structure specifically tailored for motion blurring of voxels, entitled Temporally Unstructured Volumes (TUVs). It applies an algorithm called *reves*, to produce the TUVs, which include substep motion functions for voxel data. These functions are then applied when rendering. As of yet, the procedure has only been applied to pre-rendered volume graphics, as the rendering of the frames takes a long time. [Wrenninge 2016] However, since the routine is specifically meant for time-varying volume data, there may be a way to combine it with PVM. As long as the motion functions can be precomputed, and passed to the GPU with the polynomial representations of the voxel data, they can be found within the sampling, and evaluated. For datasets with fast moving pieces of interest, this would likely

result in better looking visualizations [Wrenninge 2016]. Given that one of the primary target applications for PVM is scientific data however, motion blur may not be a desirable effect. As always, the goals must be considered.

Bibliography

Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM. doi: 10.1145/1468075.1468082. URL http://doi.acm.org/10.1145/1468075.1468082.

Mark Claypool and Kajal Claypool. Perspectives, frame rates and resolutions: It's all in the game. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 42–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-437-9. doi: 10.1145/1536513.1536530. URL http://doi.acm.org/10.1145/1536513.1536530.

Jonathan Decker. System of bound particles for interactive flow visualization. 2007. URL https://www.csee.umbc.edu/~olano/papers/theses/Decker2007.pdf.

H. Q. Dinh, N. Walker, L. F. Hodges, Chang Song, and A. Kobayashi. Evaluating the importance of multi-sensory input on memory and the sense of presence in virtual environments. In *Proceedings IEEE Virtual Reality (Cat. No. 99CB36316)*, pages 222–228, Mar 1999. doi: 10.1109/VR.1999.756955.

D Foley, Andries van Dam, John F Hughes, and Steven K Feiner. Spatial-partitioning rep-

resentations; surface detail. *Computer Graphics: Principles and Practice. the Systems Programming Series. Addison-Wesley*, 1990.

Jim Frost. How to interpret r-squared in regression analysis. 2017.

Drew Fudenberg and David M. Kreps. The visible human project. *U.S. National Library of Medicine*, 2018.

Max Gleber. Cme week: The difference between flares and cmes. September 2014.

Simon Green. Volume rendering for games. In *Game Developer's Conference*, San Francisco, CA, USA, 2005. UBM. URL `https://pdfs.semanticscholar.org/presentation/1ee5/493e437ed1d80e679b372c8de77c2b332e2b.pdf`.

M. Jin, W. B. Manchester, B. van der Holst, I. Sokolov, and R. E. Mullinix. Data constrained coronal mass ejections in a global magnetohydrodynamics model. 2017.

Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458. ACM, 1994.

Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988. ISSN 0272-1716. doi: 10.1109/38.511. URL `http://dx.doi.org/10.1109/38.511`.

A. Lu and H. W. Shen. Interactive storyboard for overall time-varying data visualization. In *2008 IEEE Pacific Visualization Symposium*, pages 143–150, March 2008. doi: 10.1109/PACIFICVIS.2008.4475470.

H. Lundstedt and P. Wintoft. Prediction of geomagnetic storms from solar wind data with the use of a neural network. *Annales Geophysicae*, 12(1):19–24, Jan 1994. ISSN

1432-0576. doi: 10.1007/s00585-994-0019-2. URL https://doi.org/10.1007/s00585-994-0019-2.

Tom Malzbender, Dan Gelb, and Hans Wolters. Polynomial texture maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 519–528, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: 10.1145/383259.383320. URL http://doi.acm.org/10.1145/383259.383320.

Feissal Damaa Mark Claypool, Kajal Claypool. The effects of frame rate and resolution on users playing first person shooter games, 2006. URL https://doi.org/10.1117/12.648609.

Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-659-5. URL http://dl.acm.org/citation.cfm?id=846276.846298.

NASA. Ccmc: Community coordinated modeling center. 2018a.

NASA. Mms: Magnetospheric multiscale mission. 2018b.

nVidia. Graphics processor specifications. 2018.

Michael Plass and Maureen Stone. Curve-fitting with piecewise parametric cubics. In *ACM SIGGRAPH computer graphics*, volume 17, pages 229–239. ACM, 1983.

H. Prautzsch, W. Boehm, and M. Paluszny. *Bézier and B-Spline Techniques*. Mathematics and Visualization. Springer Berlin Heidelberg, 2013. ISBN 9783662049198. URL https://books.google.com/books?id=fEiqCAAAQBAJ.

D. S. Reis, J. R. Stedinger, and E. S. Martins. Bayesian generalized least squares regression with application to log pearson type 3 regional skew estimation. *Water Resources Research*, 41(10). doi: 10.1029/2004WR003445. URL https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2004WR003445.

Neel P Savani. The morphology of coronal mass ejections between the sun and the earth. 2010. URL https://spiral.imperial.ac.uk/bitstream/10044/1/6040/1/Savani-NP-2010-PhD-Thesis.pdf.

L.T. Skovgaard. Applied regression analysis. 3rd edn. n. r. draper and h. smith, wiley, new york, 1998. no. of pages: xvii+706. price: £45. isbn 0-471-17082-8. *Statistics in Medicine*, 19(22):3136–3139, 1998. doi: 10.1002/1097-0258(20001130)19:22⟨3136::AID-SIM607⟩3.0.CO;2-Q.

M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. M. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V. W. Lee, A. D. Nguyen, L. Seiler, and R. Robb. Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570, Nov 2009. ISSN 1077-2626. doi: 10.1109/TVCG.2009.164.

Kazue Takahashi and Brian J. Anderson. Distribution of ulf energy (f ¡ 80 mhz) in the inner magnetosphere: A statistical analysis of ampte cce magnetic field data. *Journal of Geophysical Research: Space Physics*, 97(A7):10751–10773, 1992. doi: 10.1029/92JA00328. URL https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/92JA00328.

Alex Vlachos. Advanced vr rendering. In *Game Developers Conference*, 2015.

R Voigt. Efficient handling of time-varying fem model data in the 3d groundwater simulation system feflow. 1998.

Brooklyn Waters. Physics and frame rate: Beating motion sickness in vr. 2015.

Carsten Wenzel. Real-time atmospheric effects in games. In *ACM SIGGRAPH 2006 Courses*, pages 113–128. ACM, 2006.

Lee Alan Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. PhD thesis, University of North Carolina at Chapel Hill Chapel Hill, NC, 1991.

Jonathan Woodring and Han-Wei Shen. Chronovolumes: A direct rendering technique for visualizing time-varying data. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume Graphics*, VG '03, pages 27–34, New York, NY, USA, 2003. ACM. ISBN 1-58113-745-1. doi: 10.1145/827051.827054. URL `http://doi.acm.org/10.1145/827051.827054`.

Magnus Wrenninge. Efficient rendering of volumetric motion blur using temporally unstructured volumes. *Journal of Computer Graphics Techniques (JCGT)*, 5(1):1–34, January 2016. ISSN 2331-7418. URL `http://jcgt.org/published/0005/01/01/`.