UNIVERSITY OF CALIFORNIA, SAN DIEGO

Visualization of Time-Dependent Seismic Vector Fields With Glyphs

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Emmett McQuinn

Committee in charge:

Jean-Bernard Minster, Chair
Jürgen P. Schulze
Larry Smarr
Amit Chourasia

2010

The thesis of Emmett McQuinn is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

Chair

University of California, San Diego

2010

DEDICATION

To ∞, a very large number in my life.

EPIGRAPH

Graphical elegance is often found in simplicity of design and complexity of data.

—Edward R. Tufte

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . x

Acknowledgements . . . xi

Abstract of the Thesis . . . xii

Chapter 1 Introduction . . . 1
1.1 Introduction . . . 1
1.2 Simulations . . . 2
1.2.1 Velocity . . . 3
1.2.2 Acceleration . . . 3
1.2.3 Displacement . . . 3
1.2.4 Gradient, Divergence, and Curl . . . 3
1.3 Contributions . . . 4
1.4 Related Works . . . 5

Chapter 2 Glyph Techniques . . . 10
2.1 Geometry . . . 11
2.1.1 Square . . . 11
2.1.2 Sphere . . . 11
2.1.3 Ellipsoid . . . 13
2.1.4 Comet . . . 13
2.1.5 Twigs . . . 15
2.1.6 Triglyph . . . 17
2.1.7 Arrow . . . 17
2.2 Shading . . . 17
2.2.1 Flat Shading . . . 20
2.2.2 Diffuse Shading . . . 20
2.2.3 Halos . . . 21
2.2.4 Procedural Dipole Texturing . . . 21
2.2.5 Procedural Cross Mark Texturing . . . 21
2.2.6 Concentric Ring Texturing . . . 22

2.3 Jitter . . . 23
2.4 Displacement . . . 24
2.5 Visibility Functions . . . 24
2.5.1 Smooth Operators . . . 25
2.5.2 Bézier Editor . . . 25
2.5.3 Linear History . . . 26
2.5.4 Orientation Difference . . . 27
2.6 Scale . . . 28
2.7 Opacity . . . 29

Chapter 3 Context Techniques . . . 31
3.1 Lattice . . . 31
3.2 Isosurface . . . 33
3.3 Slice . . . 34
3.3.1 Scalar Slice . . . 34
3.3.2 Fault slip rate . . . 35
3.4 Color Jitter . . . 35
3.5 Fog Of Science . . . 37
3.6 Screen Space Light Attenuation . . . 37
3.7 Screen Space Ambient Occlusion . . . 38

Chapter 4 Hardware Systems . . . 41
4.1 Personal Computer . . . 41
4.2 Tile Display . . . 42
4.3 CAVE . . . 43

Chapter 5 Implementation . . . 45
5.1 Design Criteria . . . 45
5.2 Visualization Engine Overview . . . 45
5.2.1 Theoretical Memory Usage . . . 49
5.3 Geometry Implementation . . . 52
5.3.1 Billboard . . . 52
5.3.2 Sphere . . . 54
5.3.3 Ellipsoid . . . 55
5.3.4 Comet . . . 57
5.3.5 Twigs . . . 58
5.4 Texturing Implementation . . . 59
5.4.1 Dipole Texturing . . . 59
5.4.2 Cross Texturing . . . 60
5.4.3 Concentric Ring Texturing . . . 60
5.5 Post Effects Implementation . . . 61
5.5.1 Screen Space Ambient Occlusion . . . 62
5.6 Transparency... or lack thereof . . . 63

Chapter 6 Results and Discussion 65
6.1 Synthetic Data Use Case 65
6.2 TeraShake Data Use Case 67
6.3 Visualization for Error Purposes 71
6.4 Performance 72
6.5 Benchmarks 72
6.5.1 Benchmark Methodology 73
6.5.2 Benchmark Results 75

Chapter 7 Conclusion 85

Appendix A A Notational Note 87

Appendix B Benchmark Tabular Results 89
B.1 Quadro Engine Performance 89
B.2 GeForce Engine Performance 89
B.3 GeForce Engine Performance -O0 90
B.4 GeForce Engine Performance -O1 90
B.5 Quadro Glyph Performance for Radius 0.1 91
B.6 Quadro Glyph Performance for Radius 0.5 91
B.7 Quadro Glyph Performance for Radius 1.0 92
B.8 Quadro Glyph Performance for Radius 3.0 92
B.9 GeForce Glyph Performance for Radius 0.1 93
B.10 GeForce Glyph Performance for Radius 0.5 93
B.11 GeForce Glyph Performance for Radius 1.0 93
B.12 GeForce Glyph Performance for Radius 3.0 94
B.13 GeForce Usecase No Play 94
B.14 GeForce Usecase Play 95
B.15 Quadro Usecase Play 95
B.16 Quadro Usecase No Play 95

Bibliography 96

LIST OF FIGURES

Figure 1.1: Volumetric Rendering 5

Figure 2.1: Geometry 12
Figure 2.2: Comet Geometry 14
Figure 2.3: Pseudopipe Overlap 14
Figure 2.4: Edmond Halley Comet 15
Figure 2.5: Twigs 16
Figure 2.6: Triglyph 18
Figure 2.7: Arrow Glyph 19
Figure 2.8: Shading 22
Figure 2.9: Jittering 23
Figure 2.10: Displacement 24
Figure 2.11: Visibility Functions 30

Figure 3.1: Displacement 32
Figure 3.2: Slice 34
Figure 3.3: Color Jitter 36
Figure 3.4: Fog of Science 37
Figure 3.5: Screen Space Light Attenuation with Sparse Volume 38
Figure 3.6: Screen Space Ambient Occlusion with Seismic Vortex 39

Figure 4.1: Optiportal 42
Figure 4.2: StarCAVE 43

Figure 5.1: Visualization Engine Pipeline 46

Figure 6.1: TeraShake 2.1 Sliprate 66
Figure 6.2: TeraShake 2.1 Isosurface 66
Figure 6.3: Prior Vector Methods Compared to New Techniques 68
Figure 6.4: Error Visualization 70
Figure 6.5: Performance Increase with Optimization 75
Figure 6.6: Glyph Benchmark Laptop 76
Figure 6.7: Glyph Benchmark Workstation 77
Figure 6.8: Core Engine Benchmark 78
Figure 6.9: Shark Profile 79
Figure 6.10: Usecase Benchmark “GeForce” 80
Figure 6.11: Usecase Benchmark “Quadro” 81

LIST OF TABLES

Table 5.1: Source Line Count Complexity 48
Table 5.2: Maximum Memory Usage 51

Table 6.1: Glyph Geometry and Texturing Permutations with Vortex 84

ACKNOWLEDGEMENTS

I dedicate the pixels and ink that occupies this space to several people who helped me through my education and completion of this thesis. This work would not be possible without the help and passion of others.

The group at the Southern California Earthquake Center provided me with the initial connections and funding to pursue this research. They were able to connect me with Amit Chourasia and Jean-Bernard Minster, both domain experts in their respective fields. Jürgen Schulze, Amit, and Bernard provided a triumvirate of advice that undoubtedly contributed to this product today. It is an understatement to say this system would not exist as it stands without their help.

I would also like to thank Larry Smarr for taking time to serve on this committee an help provide some of the services at Calit2. In addition, I thank Kai Doerr and the group at GRAVITY that helped me create the CGLX codebase. Geoffrey Ely graciously assisted with comprehension, computation, and visual aesthetics. Steve Day, Kim Olsen, Tom Jordan, Rick Wagner, and Michael Norman provided useful insight, suggestions, and data that greatly influenced the focus of this work from the perspective of domain scientists. Additional thanks to the Southern California Earthquake Center and the kind staff Bob de Groot, John McRaney, and Tran Huynh. This work was funded through a sub-award USC PO#130867 to SDSC from the NSF OCI-0636438 grant as part of the SCEC ACCESS-G program.

ABSTRACT OF THE THESIS

Visualization of Time-Dependent Seismic Vector Fields With Glyphs

by

Emmett McQuinn

Master of Science in Computer Science

University of California, San Diego, 2010

Professor Jean-Bernard Minster, Chair

Seismic simulations allow us to study earthquakes in a manner not feasible with the real world. Simulations of earthquakes produce time-dependent vector fields that contain interesting geophysics. Prior visualization strategies focused on slices and volumetric rendering of scalar fields which reduces the observable phenomena. This thesis studies visualization techniques implemented in an interactive glyph visualization application called "GlyphSea" that allows scientists to explore seismic velocity fields. This work draws from a large body of work in glyph rendering and focuses on time-dependent seismic vector fields and is the result of collaboration between domain experts in visualization and seismology.

Through the study of vector visualization, several novel techniques were formed. A novel procedural dipole and cross mark texturing enhancement encodes unambiguous

vector orientation on any geometry with volume. A novel lattice method was created to show neighborhood which also enables glyph distinction. Visualization is further enhanced by using screen space ambient occlusion, jitter, halos, and displacement.

Chapter 1

Introduction

1.1 Introduction

Advances in computer hardware and software have provided scientists an alternative method to study earthquakes through simulation [CCO+08, ODM+06, ODM+08, TYRg+06]. Several research groups develop computational models to simulate potential earthquake scenarios. These simulations use several parameters like magnitude of the fault rupture, dynamics of fault slip along the fault, and ground characteristics that specify how a body of earth reacts to a seismic wave.

The output of simulations produce an enormous amount of temporal volumetric information. Many data products are further derived from simulation outputs and these could include seismograms of ground velocity, displacement, cumulative peak velocity, curl, and divergence. The analysis of this data is key to understand seismic phenomena. Some features of interest include regions undergoing severe shaking, source directivity effects, and wave guide effects. Surface scalar maps and plots have traditionally been used by scientists to represent simulation data. With advent of computational simulations, now scientists have access to vast amounts of volumetric information that could lead to better understanding of the three dimensional ground motions.

Visualization is key to comprehension of massive simulations. Recent studies in visualization of earthquake simulations have largely focused on scalar visualization. These have included 2D maps or 3D volume renderings. There have also been attempts to visualize vector quantities with 2D LIC and particle advection systems. While these

visualizations can provide an overview of data, they fall short in providing rich and deep scientific insights.

Working in close collaboration with seismologists we determined that the existing techniques are not sufficient for rigorous seismic data exploration. Thus the motivation for this work is to build a novel application called "GlyphSea" which allows interactive exploration of temporal and multi-field seismic data. GlyphSea provides an interactive temporal volumetric vector environment that surpasses prior batch scalar visualiations. Displaying higher a dimentional system enables scientists to combine several aspects of earthquake simulation into a comprehensive visual representation. GlyphSea's main visualization primitive is a glyph which encodes both magnitude and direction of a vector data either from velocity, displacement, or acceleration information. GlyphSea is further enriched with the addition of contextual information like fault sliprate, geographic context, and isosurface of ground stiffness.

Current graphics hardware is capable of displaying a very large number of glyphs at interactive rates. However the choice of glyph techniques including shape, shading, coloring, and transfer function are key issues for insightful interactive exploration. This study presents a quiver of techniques which utilize glyph shape, texture, position, scale, and context enrichment for interactive exploration of multi-field temporal seismic data.

1.2 Simulations

Simulations today are run from ten to several hundred thousand cores producing petabytes of data. Thew AWM-ODC simulation used by many southern california simulations employs a finite difference method to solve for the wave equation over many timesteps. The initial conditions are what define the simulation, but the unknown is the behavior of the system. Comprehension of the dynamics of the simulation can be enhanced through visualization, which is the motivation for this work. Some of the initial conditions, such as the ground stiffness volume, can enhance comprehension. On the other hand, the scalar and vector fields produced provide quantities useful for comprehension. These same fields are used for visualization and are described in the following subsections.

1.2.1 Velocity

Velocity is the physical quantity commonly output as the result of simulations. Large velocity magnitude indicates strong ground motion, which provides significant effects for us surface dwellers. While magnitude may be useful as a guide to seismic hazard, orientation may be too specific for hazard analysis. However, velocity orientation is useful in studying simulations and the geophysical effects within.

1.2.2 Acceleration

Acceleration is the derivative of velocity with respect to time. High magnitude acceleration can be a useful visualization parameter because this shows the areas where velocity is changing the most. The areas that have strong change in velocity magnitude provide intuition about wavefronts, which is useful for tracking wave propagation throughout a volume.

1.2.3 Displacement

Displacement is the integral of velocity with respect to time. Visualizing displacement can give an impression of how the earth might be moved for a simulation. The areas with the largest displacement are generally adjacent to the fault rupture, due to energy decaying approximately $1/distance^2$.

1.2.4 Gradient, Divergence, and Curl

The gradient $\nabla \mathbf{v}$ of a vector field describes the direction which the field increases the most, while the magnitude describes the rate of increase. It is calculated with a partial derivative of the spatial dimensions of a field $\mathbf{v}$:

$$\nabla \mathbf{v} = \left\langle \frac{\partial \mathbf{v}}{\partial x}, \frac{\partial \mathbf{v}}{\partial y}, \frac{\partial \mathbf{v}}{\partial z} \right\rangle$$

The gradient is used for calculating curl and divergence of a field. While the gradient may give some insight into wave motion, curl and divergence give additional

insight into the field. Divergence is calculated simply with a dot product $\nabla \cdot \mathbf{v}$. Divergence gives insight into compression and expansion of waves within a volume. On the other hand, curl can give insight into vorticity in a volume. Vorticity describes the amount of rotation around a point in a vector vector field. Curl describes the velocity field. Curl is defined as $\nabla \times \mathbf{v}$, where ∇ is the gradient of the velocity field and $\mathbf{v}$ is the velocity vector.

Because gradient, curl, and divergence are applied to the spatial rather than temporal field, they they are time independent and can be performed with poor temporal resolution simulations. This is relevant for large seismic simulations where every timestep is not stored to disk. Computation along the surface of the volume is poorly defined due to boundary conditions, so visualization and computation must incorporate this.

While divergence is a scalar quantity, it can still be visualized with GlyphSea. On the other hand, gradient and curl are vector quantities which can be displayed just as easily with GlyphSea as a regular velocity vector volume.

1.3 Contributions

The following are the key contributions of this research:

- Exploration of visualization techniques for improving physical intuition of time-dependent vector seismic volumes.
- A novel technique to encode and display orientation information of vector data by using procedural dipole and cross mark texturing.
- A novel method for enhancing neighborhood distinction of glyphs by using kelp-lattice and full-lattice.
- First use of screen space ambient occlusion with glyphs in a post effects pipeline. This enables us to enhance depth perception without degrading performance as this method is not dependent on the scene complexity but rather on screen resolution.

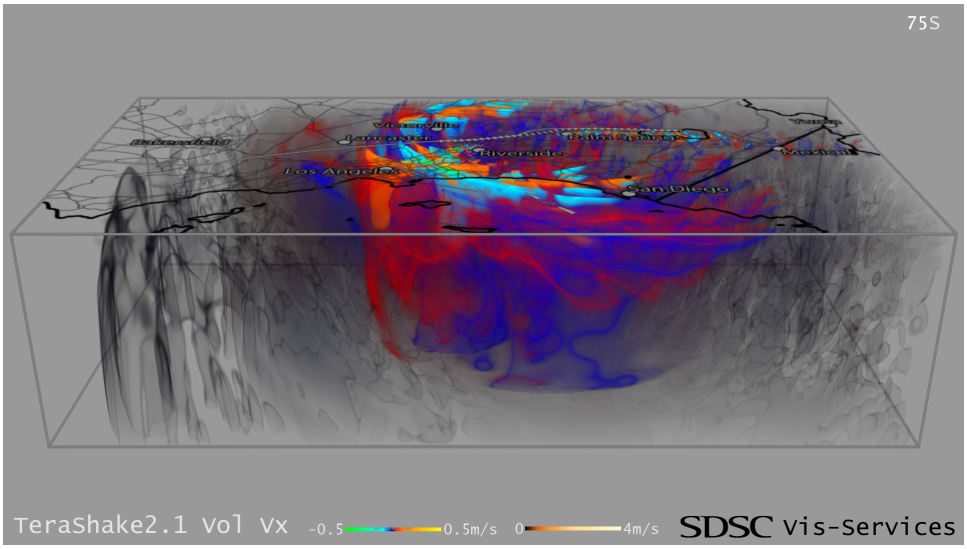


Figure 1.1: A volumetric rendering has occlusion problems even with transparency. This volume looks at only the x-component of velocity. This image was produced by Amit Chourasia of SDSC and used with permission.

- First to employ vector glyph geometry for full volume visualization of seismic simulations.
- Providing contextual information like geographic map, isosurface of ground characteristics, and fault sliprate for rich exploration of seismic data.

1.4 Related Works

Seismic visualization is a moderately studied topic Much of existing work is centered on visualization of scalar fields using direct volume rendering [CCC+07, CCO+08, TYRg+06] or isosurfaces. These methods have been explored in great detail and are useful for showing scalar quantities or wave fronts. Simply depicting scalar quantities of an intrinsically vector field loses information relevant to science.

There are several prior methods used to visualize seismic vector fields. The IEEE Vis 2006 contest had several entries that proposed vector visualization. A novel method was proposed by Bürger et al. [KBKW07] to visualize vector quantities using particle advection and focus+context techniques. Particle advection techniques lack temporal

succinctness that is useful for large time dependent seismic volumes. For each frame of simulation, many frames must be rendered with particle advection to determine differences in the vector field. Furthermore, it is difficult to register vector field differences when the temporal resolution of the simulation is poor. Particle advection will not give good insight into 3D surface slices, because often vectors point outside of the volume and the particle will simply reset. This is unfortunately common with seismic simulations due to the difficult data management problem of computing and storing many high resolution timesteps. To alleviate some of these concerns, Bürger et al. used colored glyphs to imply the vector orientation, where the red, green, and blue colors were combined in proportion to velocity components along X, Y, and Z axes respectively. While this is a succinct method to encode some vector information, it discards vector magnitude and sign in addition to being difficult to interpret. With an ideal perspectively linear colorspace, a user must train to determine color and axis correlation, and train for color blending characteristics. Deciphering orientation based on colors is highly unintuitive except for base cases. This method is convenient because it is visually succinct, where a singe pixel can represent a vector quantity.

Although the XYZ/RGB method is succinct, it is not intuitive. LIC is a vector field visualization technique that is visually succinct but more intuitive. The study of 2D LIC applied to seismic simulations was the focus of a Masters Thesis by Nima Shamlo [Sha05]. One issue is that although LIC provides indication of flow, it does not give information at discrete points and gives no information to sign of a vector. A problem exposed by this method, and seen in other studies, is that although LIC is useful in 2D fields, it is extremely difficult to get right in 3D circumstances.

LIC in 2D can be very useful for visualizing flow even in 3D cases when most of the motion is in the 2D LIC plane. However, even in these ideal circumstances, it is difficult to represent both magnitude and sign for the velocity field. The best scenario would be to use a bivariate colormap for negative magnitudes and positive magnitudes. An instance of using 2D LIC without magnitude or sign information was done by Yu et al. [YMW04] using 2D LIC with seismic volumes to provide orientation insight for surface planes in combination with scalar volumetric visualization. Recently, Chen et. al [CCM09] have proposed volumetric enhancements with deformable textures for seismic

volumes. We chose to implement a different technique using glyphs to visualize discrete vector quantities.

Glyph-based rendering is used extensively in scientific visualization. One of the recent uses of glyphs is to represent DT-MRI tensor fields, with interesting visualizations by Westin et al. [WMM+02] and Bergmann et al. [BKLW06]. Glyphs have also been applied within the context of seismic visualization. Nayak et al. [NLK+03] use glyphs to display discrete points on the ground surface from real time seismic sensor data. This method used several displays with one display per glyph to represent surface tensors from the field, but it lacks the density necessary for full volume visualization. For the full volume visualization, Neeman et al. [NJP05] applied plane-in-a-box tensor glyphs to represent stress for a single timestep of a geomechanics simulation.

Glyph geometry is a well studied topic, and there exist many geometries to use. We chose four basic glyph types: squares, spheres, ellipsoids, and comets. Gordon Kindlmann [Kin04] showed that superquadratic glyphs are advantageous due to decreased perceptual ambiguity. However they increase implementation complexity, degrade interactive performance, and do not completely resolve view dependence problems. Gumhold [Gum03] suggests a clever implementation of ellipsoid splatting, where it can be implemented using a simple ellipsoid intersection test by translating to spherical coordinates, while this is not the general case for superquadratic glyphs. The comet glyph was developed independently, but similar glyphs have been created in many scenarios. Perhaps the earliest such use is by Edmond Halley describing the 2D velocity field of ocean winds in 1686 [Hal86]. More recently, Guthe et al. [GGS01] employed a comet glyph in 3D. We chose these four glyphs due to their compactness, simplicity, familiarity and implementation ease. While more complex glyph geometries are possible, such as an arrow, they are difficult to implement with a fast procedural method necessary for large volume visualizations.

The directional component of vectors is often simply drawn using glyph orientation. However, additional cues can be useful to resolve sign ambiguity and further enhance orientation. Gordon Kindlmann [KW99] used a barycentric colormap transformation to provide orientation enhancements for spherical glyphs. The complex shading model proposed in this study makes it useful for limited cases.

Mesh regularity with glyphs can introduce distracting visualization artifacts such as moiré patterns. One method used to compensate for regularity is to modify the density of glyphs in a region. This can either be done by automated means or interactively, through use of a particle probe. Kindlemann [KW06] explored the concept of glyph packing, which can compensate for both regularization artifacts and occlusion, while Kondrativa explored a particle probe for seeding glyphs [KKW05]. Another method to compensate for regularity is using stochastic jittering, which was compared to other 2D methods in a user study by Laidlaw et al. [LKJ$^+$05].

Various illustrative context techniques have been explored to enhance glyph geometry. Guthe et al. [GGS02] proposed halos to distinguish glyphs from one another and appear distinct. Gribble et al. [GP06] demonstrate advanced lighting models that incorporate shadows, ambient occlusion, and diffuse inter-reflection that provide additional depth cues. Everts et al. [EBRI09] describe depth-dependent halos to enhance line geometry with depth cues based on line width. Luft et al. [LCD06] suggest a method of illustrative depth techniques by unsharp masking the depth buffer, which is the basis for a cheap screen space ambient occlusion method (SSAO). Screen space ambient occlusion is able to provide advantages of depth-dependent halos [EBRI09] with a runtime dependent on pixels rather than scene geometry that is important for dense seismic volumes which have a large number of glyphs.

Several computer science advances were necessary to display glyphs interactively. With the introduction of the programmable graphics pipeline on commodity hardware, it is feasible to look at volumes interactively using particle and glyph systems on a typical workstation. There have been several advances in realtime graphics programming that allow for an interactive glyph visualization. One advance has been GPU-based particle systems, which allows for particle advection techniques to happen in real time. Two groups simultaneously created GPU-based particle systems that reduce memory copies from a GPU to a CPU and leverages the GPU's expansive vector processing capabilities [KSW04][KLRS04]. Another advancement comes from ellipsoid glyph splatting in pixel shaders. Stefan Gumhold [Gum03] discovered a way to project ellipsoids into a parametric space which eases computation in the fragment shader. This technique is discussed further in section 5.3.3. This work was improved upon by Sigg

et al. to apply to pointsprites with halos in [SWBG06]. The combination of splatted ellipsoids and particle advection systems was explored by Kondratieva et al. [KKW05] with DT-MRI data.

Chapter 2

Glyph Techniques

Glyphs offer an easy way to represent vector fields where each glyph is positioned and oriented in vector direction. Color, scale, and opacity can be applied to glyphs to represent velocity magnitude. Glyphs work well with sparse datasets where they are able to occupy a large imagespace without occlusion, but for dense regular data glyphs can become cluttered and difficult to interpret. Seismic simulations often have dense regular grids of vector information. Several methods were employed to alleviate imagespace clutter such as high resolution displays, interactive camera exploration, and visibility functions that emphasize regions of interest.

There are three main components to a glyph: geometry, texture, and context. Ideal geometry can display the orientation of a glyph intuitively using the fewest pixels possible. An ideal texture can enhance geometry orientation and help reduce the number of pixels needed to represent a glyph. Contextual information is useful to convey information about societies of glyphs and the data they are representing.

Due to advances in computing power, each glyph is dynamically generated on a billboard rather than using geometry or lookup textures. This also makes the glyphs resolution independent. Modern graphics cards can execute more than 14 operations for every texture lookup [AMHH08], while the shader code for procedural dipole glyph generation generate 25 to 58 fragment shader operations.

There are several disadvantages to using glyphs. Glyphs often have view-dependent geometric ambiguity, difficulty with occlusion in volumes, and require significant investment of imagespace compared to pixel or voxel imagery. These disadvantages motivated

several design decisions and focus for research. Through visibility functions, interactive camera movement, texutring techniques, and high resolution displays the disadvantages have been mitigated.

2.1 Geometry

Glyph geometry is one of the core techniques to encode data in visual form. Related literature shows three common glyphs: arrows, spheres, and ellipsoids. These basic glyph geometries were explored in addition to some improvisation with comets and triglyphs. GlyphSea provides four basic glyph geometries that can be interactively switched to see features of interest. A comparison of different geometries is shown in Table 6.1.

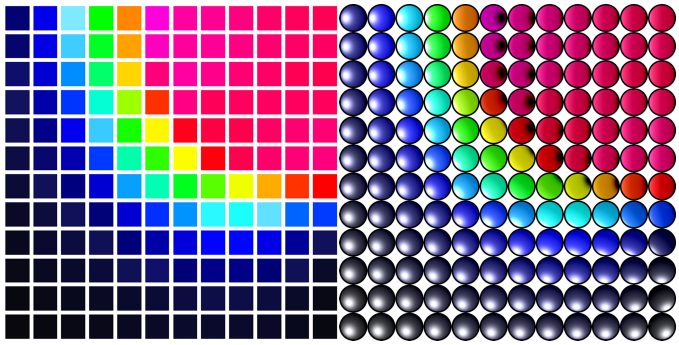
2.1.1 Square

The glyph geometry is a square billboard with a solid uniform color which is depicted in Figure 2.1a. This representation can be useful to provide a "background" of understanding for color mapping in a volume. The square glyph is not used for representing vector information.

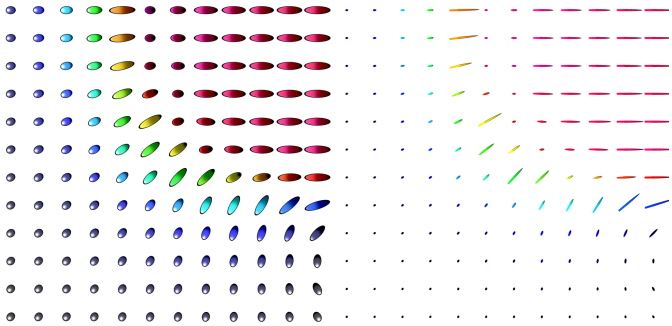
2.1.2 Sphere

The glyph geometry for the sphere glyph is a sphere as seen in Figure 2.1b. Sphere geometry inherently does not provide an indication of orientation due to rotational symmetry.

The projection of a sphere can be shown to be rotationally symmetric as follows. A sphere is defined using spherical coordinates $\theta, \phi, r = 1$ and can be represented in to Cartesian coordinates as $\langle x, y, z \rangle = r\langle \cos(\theta)\sin(\phi), \sin(\theta)\sin(\phi), \cos(\phi) \rangle$. The silhouette of the sphere can be found in imagespace by setting $z = \cos(\phi) = 0$. By solving for $\phi = \arccos(0)$ and substituting $\sin(\phi) = 1$, the trace of the silhouette in imagespace is defined as $\langle \cos(\theta), \sin(\theta) \rangle$.



(a) Square (b) Sphere



(c) Ellipse (d) Comet

Figure 2.1: Figure showing four glyph geometry types. Panel (a) shows the square planar glyph with a uniform color. Panels (b,c,d) show the sphere glyph, ellipsoid glyph, and comet glyph respectively which also indicate the direction using dipole texturing. Notice the orientation makes the ellipsoid, and comet glyph appear foreshortened when the glyphs are not parallel to the projection plane. This foreshortening will take place for any elongated glyphs, including simple lines and arrows.

If an object is invariant under rotation, then it is rotationally symmetric. Furthermore, the projected silhouette can be shown to be identical across all rotations. Given a rotation for a sphere with angles $\theta', \phi' \in \mathbb{R}$, then the transformed spherical coordinates will become $\theta_R = \theta + \theta'$, $\phi_R = \phi + \phi'$. The silhouette of the sphere in imagespace is then be defined by $\langle \cos(\theta_R), \sin(\theta_R) \rangle$ which produces the same projected silhouette from before transformation.

Despite rotationally symmetry, a sphere can encode and display orientation information with dipole texturing as seen in Figure 2.1b. Spheres are preferred in situations where the pixel to glyph ratio is low and where it could be problematic to interpret data with ellipsoid and comet glyphs. This is because ellipsoid and comet glyphs project to small spheres when vector direction is into or away from the camera.

2.1.3 Ellipsoid

The ellipsoid glyph is created by using the direction of a vector to correlate with the principal axis of the ellipsoid. The radii components for the three different axes are fixed to produce a uniform geometry see (Figure 2.1c). Ellipsoid geometry can provide an orientation in cases where one principal axis is distinguishably larger than the other two axes, but in other cases it is still ambiguous. This glyph is very common with tensor imaging, and the principal axes directions are aligned with the eigenvectors of the tensor. Ellipsoids require extra computation compared to a sphere glyph, but they may provide better understanding of the data.

2.1.4 Comet

The comet glyph is a skinny ellipsoid with one end tip removed, see (Figure 2.1d). This glyph was designed to emphasize the "flow" and is capable of showing flow in volumes with a large glyph density. The comet glyph went through several iterations to come with the current geometry and shading.

Glyph density is ordinarily a concern because greater density increases the likelihood that there is glyph occlusion within a field. However, in some fields neighboring glyphs contain similar vectors, and good visualization can clearly show this "flow".

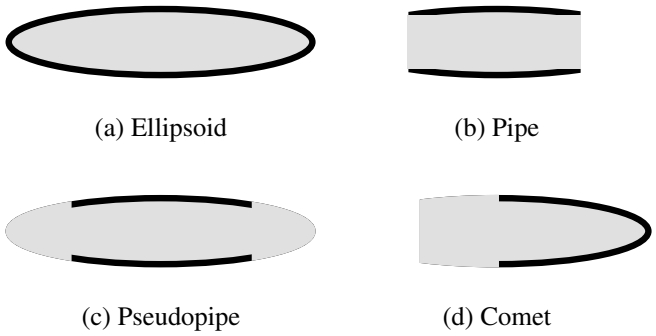


Figure 2.2: Figure comparing several iterations of glyphs created to enhance flow. Starting with the ellipsoid primitive (Figure 2.2a), the caps were removed (Figure 2.2b), then just the halo (Figure 2.2c), finally resulting in the comet (Figure 2.2d).

Figure 2.3: The pseudopipe geometry and shading was designed to enhance flow when glyphs overlap.

Rather than avoid glyph occlusion, the comet glyph was designed to leverage glyph overlap as a method to give some continuity of flow.

The first method is the "pipe" (Figure 2.2b). This is an ellipse with the tips removed. For comparison, a glyph with no effects was created. This does not have distractions nor provide the wrong intuition, but it is easy to have regions that are unintelligible.

The second method is the "pseudo pipe" (Figure 2.2b). An edge is drawn on the center of a stretched ellipse and the caps have no edge. The idea is that when two glyphs overlap, the caps have no edge and should direct the viewer along the glyphs as seen in Figure 2.3.

The last method was derived from both methods. It implements an ellipsoid tail and a pseudo pipe head, which is labeled as a "comet" (Figure 2.2d). Combined with proper shading, it uses geometry and luminosity to provide a strong indication of flow. Similar glyphs have been developed previously. Perhaps the earliest was done by Edmond Halley [Hal86], where he used the thickness of a line to enhance the vector

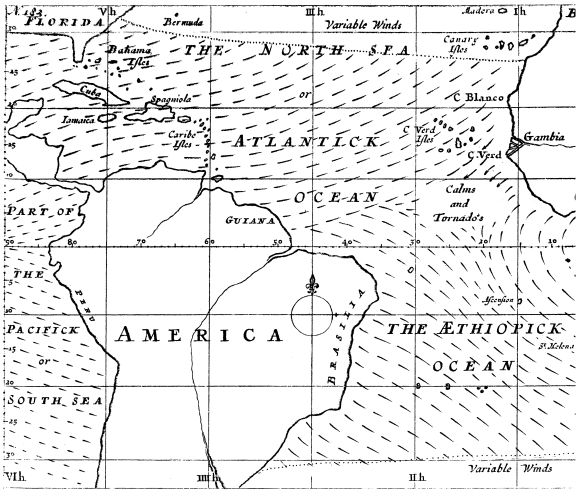


Figure 2.4: Edmond Halley utilized a comet like glyph to represent tradewind directions in his 1686 seminal paper on trade winds in *Philosophical Transactions* [Hal86].

direction of tradewinds, as seen in Figure 2.4.

An informal study was performed comparing the different comet rendering techniques with the same data and given to several scientists, where it was determined that the comet glyph was preferable. The comet glyph was chosen with the current shading method due to this study.

The comet glyph works very well in indicating direction where the glyph orientation is not perpendicular to the viewing plane. In regions where the comet glyphs are nearly perpendicular to the viewing plane, it becomes hard to estimate the magnitude and direction because the ellipsoid's cross section appears as a circle. This could also lead to the worst case where the circle is smaller than a pixel. One way to mitigate projection problems is to interactively change the viewing position.

2.1.5 Twigs

A problem with glyphs is that they require many pixels to represent, which increases occlusion with volumes. A minimal glyph is an oriented line. Orientation of the line gives the direction of the vector, while line length represents the magnitude. This line glyph is called a "twig" and can be seen in Figure 2.5.

Although this glyph initially showed promise, glyph distinction is a significant

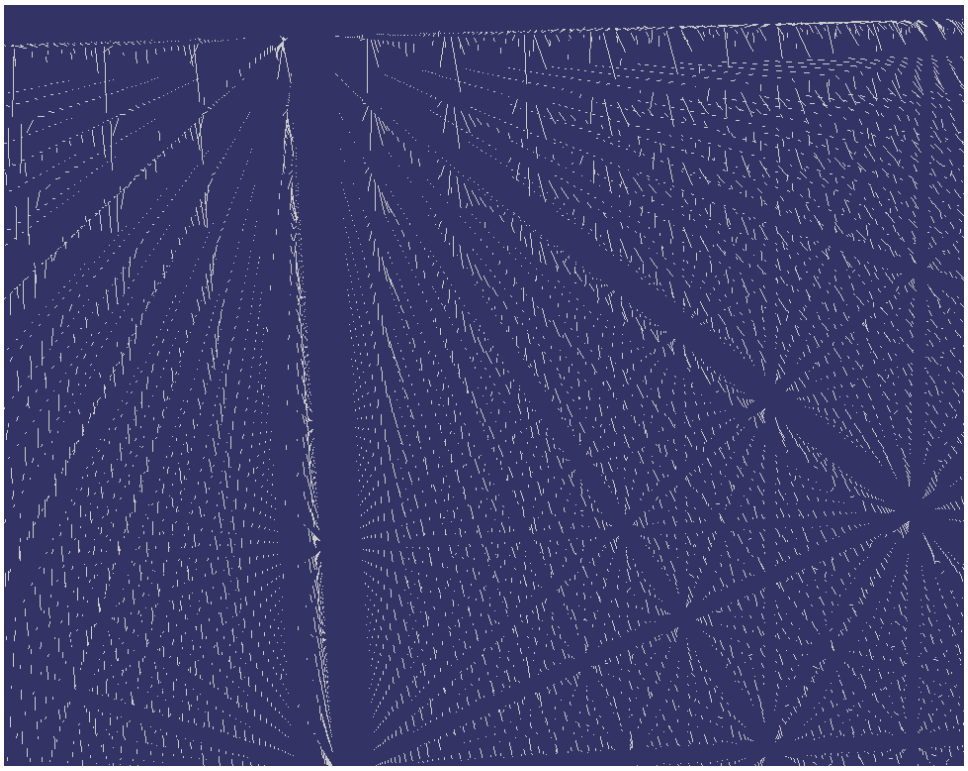


Figure 2.5: Twigs representing a vector field. Twigs reduce occlusion but become cluttered as a result.

problem that was unresolved. The lines are difficult to resolve individually which causes interpretation problems with overlapping glyphs.

2.1.6 Triglyph

The triglyph was created in an attempt to resolve difficulties with twigs. It is created as a simple oriented geometry with two triangles as caps and the body filled between (see Figure 2.6). The advantage over a line is that the body contains volume, such that glyphs can be more readily distinguished. Furthermore, the glyphs were drawn with cel-shading to provide halos which helps with glyph distinction.

2.1.7 Arrow

The arrow is a commonly used glyph to encode and display vectors. However, the arrow glyph has several problems that make it not ideal. The arrow glyph is not view-independent. Arrows require a large imagespace, so using them for dense volumetric representation causes too much clutter and occlusion (see Figure 2.7). Additionally, arrows require a complex geometry. A large number of 3D arrows can easily overwhelm the GPU and affect interactive performance. Simply loading geometry is not resolution-independent and would require a level of detail tessellation scheme to preserve high quality at large resolutions. Because of these issues arrows have not been implemented. While it is possible to raytrace arrows procedurally with a basic cone and cylinder primitives, these tests would be more involved than a ray-sphere or ray-ellipsoid test and negatively impact performance.

2.2 Shading

Glyph shading can play an important role in observing features of interest. A combination of texture and lighting can provide useful shading that enhances glyph comprehension. All shading methods are procedurally generated to be resolution independent, which proves useful on large resolution displays and workstations alike.

Shading methods can enhance glyph distinction and glyph geometry. A variety

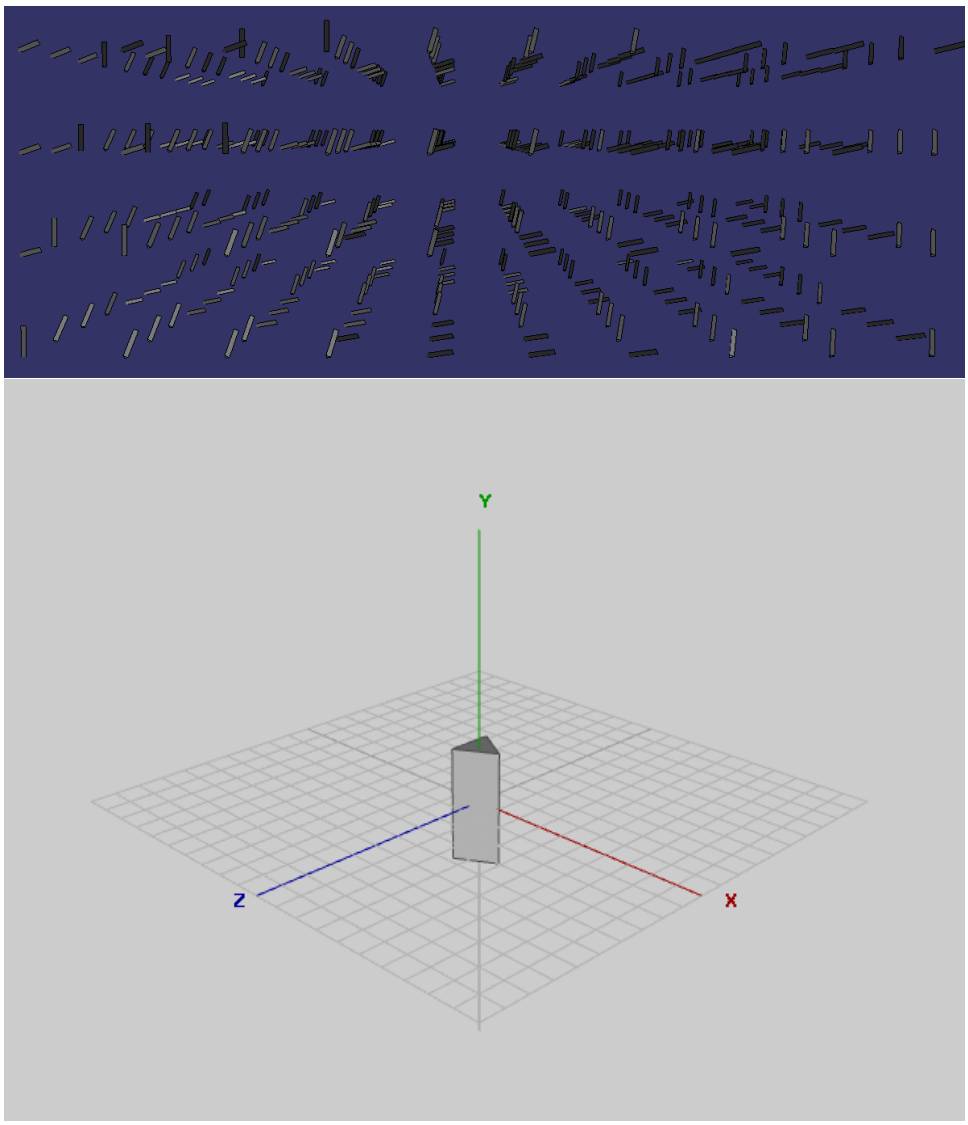


Figure 2.6: An individual triglyph (below) and triglyphs in a field with cel-shading (above).

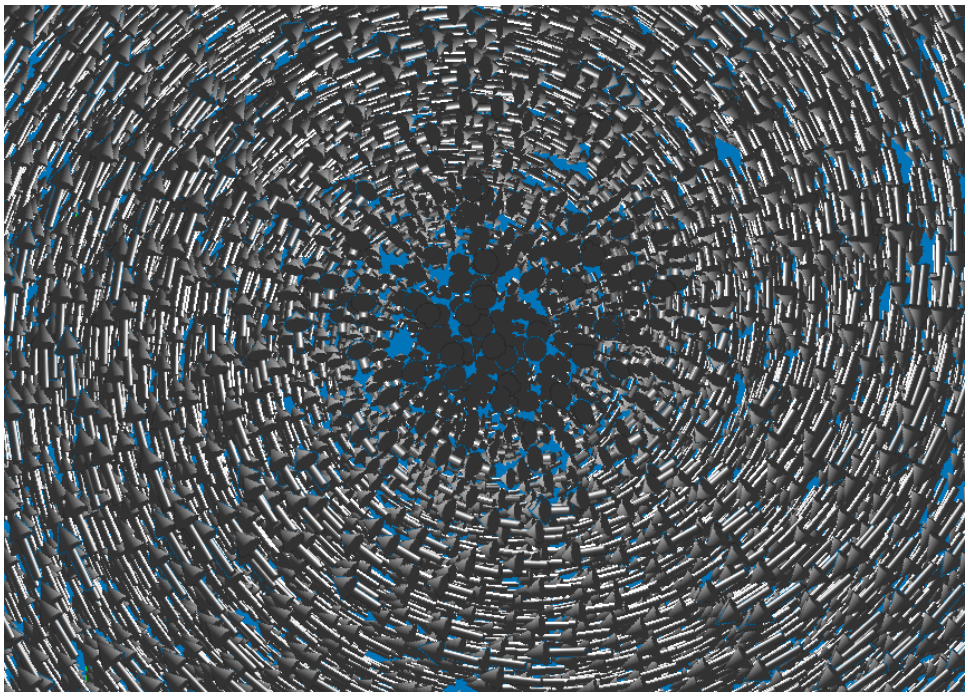


Figure 2.7: The 3D arrow is a common glyph used in vector visualization but provides extra clutter that causes distraction from interpretation. This particular instance depicts flow into a drain. Notice that the geometry is ambiguous as the vectors begin to point down into the drain. This image was produced using the tum.3d Particle Engine available online (`http://wwwcg.in.tum.de/Download/PE`).

of shading derived from common systems were used as the foundation to improvise procedural glyph texturing to enhance orientation and eliminate sign ambiguity of vectors. These features are all created dynamically within pixel shaders rather than with traditional texture sampling. A comparison of different texture methods is shown in Table 6.1.

2.2.1 Flat Shading

In flat shading, a glyph is drawn a solid uniform color (Figure 2.8a). This method simply sets the whole color of a glyph to the same color, and does not apply diffuse shading or directional enhancements like dipole texturing. It is extremely important to have halos or ambient occlusion with flat shading to provide glyph distinction and glyph depth cues.

2.2.2 Diffuse Shading

Diffuse shading is a simple technique which can provide surface shape context. The traditional diffuse lighting equation for realtime graphics is based on a Lambertian reflectance model, providing an intensity I_d given a surface normal vector $\mathbf{n}$, incoming light vector $\mathbf{l}$, and diffuse surface constant K_d:

$$I_d = K_d * \mathbf{n} \cdot \mathbf{l}$$

Although other light models exist that describe a large array of surfaces using microfacets, this simple equation is sufficient to provide surface cues. If a surface is flat, the surface normals will all be in the same direction leading to similar lighting across the surface. If the surface is continuous and not flat, then there must be some surface normals that have different orientations. These differences in surface normals are what provides some cues of surface shape. Diffuse lighting is used with isosurfaces and provides the foundation for the dipole texturing method.

2.2.3 Halos

Halos provide a dark outline around glyph edges. This glyph outlining allows the viewer to distinguish between glyphs adjacent to each other. Glyph overlap causes difficulty with depth perception of the volume. Improving distinction of overlapping glyphs allows a large number of glyphs to be displayed. This is often necessary for volumetric visualization in limited display area.

Halos can either be created with a soft edge or hard edge. A hard edge simply has a uniform edge color for the stencil drawn around the glyph. The soft edge provides a smooth transition from the glyph's color and the edge color. Both were tried and the hard edge seemed to provide better glyph distinction. However, soft edges are used with the comet glyph where glyph distinction is not the motivating feature.

2.2.4 Procedural Dipole Texturing

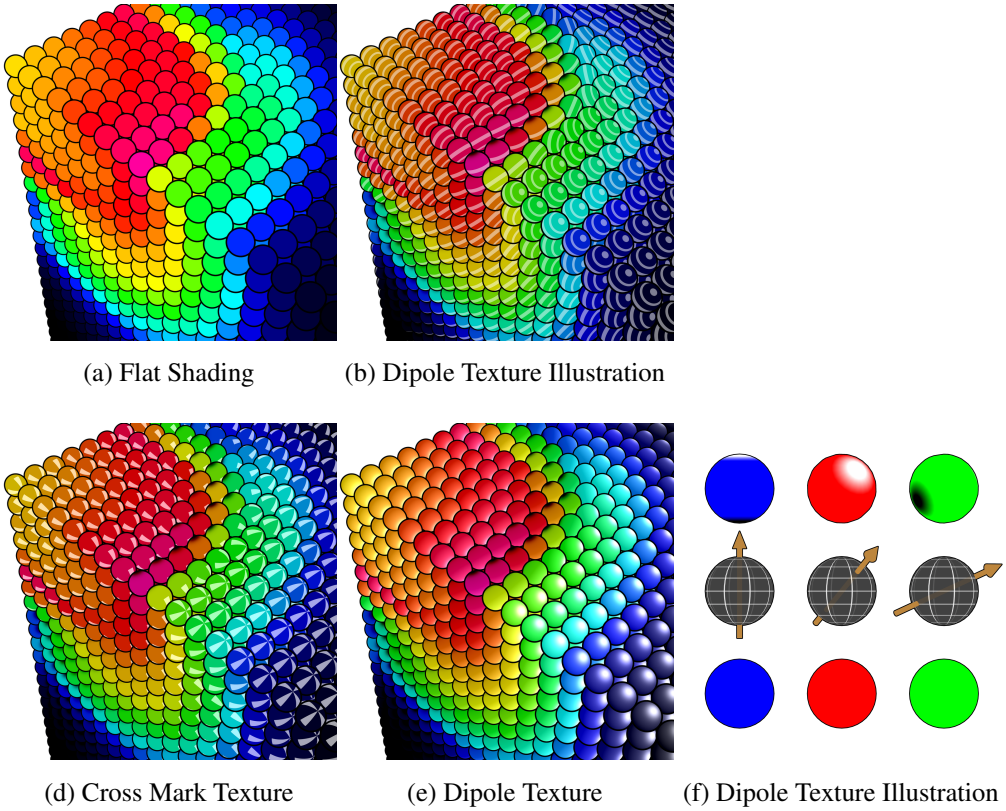
Dipole texturing (Figure 2.8e) is a novel method that encodes orientation information of a vector by procedurally painting light and dark spots on opposite ends. The lighter spot indicates the heading direction of the vector, whereas the darker spot is the antipodal point indicating the tailing end. This technique enables the viewer to unambiguously identify the vector direction independent of the viewing angle, as the viewer can see either a light spot or dark spot or both. Applying procedural dipole texturing to sphere, ellipsoid and comet geometry enables direction disambiguation. Furthermore, this technique can be extended to other geometries.

2.2.5 Procedural Cross Mark Texturing

Cross Mark Texturing (Figure 2.8d) is a novel method where four converging triangles on one end of the glyph create a cross mark like texture, while the opposite end has a dark painted spot (similar to dipole texture) that indicates vector orientation. In situations where the dipole texture saturates the display with bright and dark spots the cross mark can be preferable.

2.2.6 Concentric Ring Texturing

Concentric Ring Texturing is a novel method where several rings are drawn from the equator of a glyph to a pole. While concentric rings (Figure 2.8b) provide information about direction, the apparent orientation is orthogonal to the actual direction of the vector. This texturing is still useful in some cases, such as indicating the center of a vortex, but the viewer needs to remember that the orientation is along the polar axis and not along the diameter.

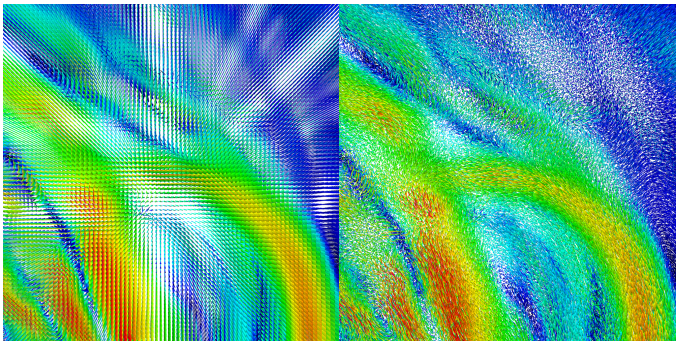


(a) Flat Shading (b) Dipole Texture Illustration

(d) Cross Mark Texture (e) Dipole Texture (f) Dipole Texture Illustration

Figure 2.8: The images show the four texturing methods (a, b, c, d) with outline halos applied to sphere glyphs. Panel (a) shows flat texturing, panel (b) displays concentric ring texturing, panel (c) shows cross mark texturing centered on the pole to indicate direction of the vector, and panel (d) shows dipole texturing with opposite bright and dark spots to indicate vector direction. Panel (f) displays how dipole texturing is able to show vector quantities with a rotationally symmetric geometry.

2.3 Jitter

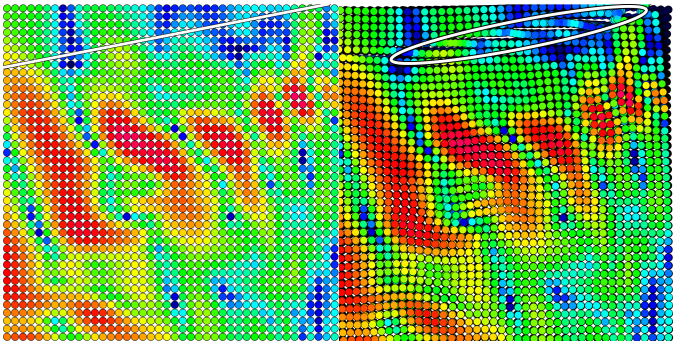


(a) No Jitter (b) Jitter

Figure 2.9: The left panel shows the moiré patterns resulting from the uniform regularity of the grid. The distraction caused by these patterns is mitigated by jittering the position of the glyphs by a pseudo-random amount that can be interactively tuned, see right panel.

Seismic data is commonly uniformly spaced on a rectilinear grid. When glyphs are placed on the regular grid they create moiré patterns (see Figure 2.9) in perspective view, which distracts the viewer from focusing on regions of interest. This problem is further exacerbated with stereo displays.

To compensate for uniform regularity, moiré effects can be reduced with stochastic jitter. Laidlaw et al. [LKJ+05] created a user study with 2D glyphs that indicates a jittered field improves comprehension over a regular grid. In 3D, projective distortion creates moiré patterns. To reduce this distraction, we offset the glyph position with a pseudo-random amount such that the particles remain in their voxel region. This stochastic jittering method drastically reduces the moiré patterns. In addition, jitter probabilistically reduces occlusion of glyphs stacked in columns orthogonal to the viewing plane. GlyphSea allows users to customize jittering distance interactively as the results vary based on the type of glyph, size of the glyphs, and underlying data. Observation finds that a jittering distance less than a half grid distance from the center of the glyph is sufficient to eliminate moiré.



(a) No Displacement (b) Displacement 10x

Figure 2.10: The left panel (a) shows the result when the glyphs are shown in their grid location, the slanted white line on the top indicates the earthquake fault. The right panel (b) the glyphs are displaced by 10 times the actual displacement which reveals the shears in the fault (see circled region).

2.4 Displacement

Although the structure of uniform spacing creates moiré patterns, the structure can also be leveraged to display physical ground displacement with glyph positions (see Figure 2.10). GlyphSea provides an interactive interface to exaggerate the displacement which allows the user to easily view where the glyphs have moved. The exaggeration is necessary as the actual amount of displacement is often several orders of magnitude smaller than the dimensions of the full volume.

2.5 Visibility Functions

Often the interesting features in volumetric data are located beyond the surface. This poses a problem where glyph occlusion prevents the view of interior glyphs. GlyphSea provides interactive filters to carve out features of interest from volumetric data. These filters can be flexibly changed to 3 quantities: the input field, the temporal derivative of the field, or the temporal integral of the field. With a typical earthquake simulation, this would be velocity, acceleration, or displacement. The end result can effect visibility by either scaling the glyph or changing the transparency of a glyph.

2.5.1 Smooth Operators

There are several common operators that produce a smooth interactive experience. These are similar to value distributions within a field: linear, power, logarithmic, exponential, and uniform. While the visibility function for regular seismic fields is either the magnitude of acceleration, displacement, or velocity, the different operators can be interchanged flexibly to showcase the underlying data. In addition to changing the operator, a scalar value s is used to exaggerate the various functions:

- linear = $s * ||\mathbf{v}||$
- power = $||\mathbf{v}||^s$
- exponential = $e^{s*||\mathbf{v}||}$
- logarithmic = $\log(s * ||\mathbf{v}|| + 1)$ [1]
- uniform = 1

In seismic datasets an ideal scale function would mask out the uninteresting regions, namely those without wave motion. Logarithmic and linear scale are typically the most useful for seismic datasets. Uniform scale can be useful when initially looking for features with large dynamic range. A comparison of the different functions can be seen in Figure 2.11.

2.5.2 Bézier Editor

The Bézier curve editor presents an interface where a visibility function can be explicitly sketched. This provides the greatest flexibility and control by a scientist. This can be useful for complex dynamic ranges often exhibited in simulations. The y-axis represents the normalized visibility scalar (in the range 0 to 1), while the x-axis represents the normalized field magnitude. This allows a scientist to sketch a visibility function that fits the data precisely rather than using a standard function like power, exponential, or linear.

[1] Logarithmic scaling is incremented by 1 to shift for the case where $||\mathbf{v}|| = 0$ as the value $\log 0$ is undefined.

2.5.3 Linear History

The human brain seems to focus on areas that are unpredictable while maintaining persistence of vision for the areas that it has already recognized. The linear history method was created in an attempt to extend this philosophy and provide an intuitive mean for volumetric visibility. In its essence, the linear history method attempts to hide "predictable" areas which the brain would hopefully have already processed and realized their function. Only novel "unpredictable" areas will be given maximal visibility.

Put simply, the Linear History method was created from the principle that areas that are predictable are not of significant interest compared to startling and unpredictable areas. A very simple algorithm was constructed that uses a linear prediction model. This algorithm uses linear extrapolation to guess the value of the current timestep based on the prior two timesteps. Given the prior two timesteps t_1, t_0, and the current timestep t_2, the expected timestep t_2' is calculated using the line equation $y = m * x + b$, where $x = 2$ because we are extrapolating based on the two prior timesteps:

$$m = \frac{t_1 - t_0}{0 - 1}$$

$$b = t_1$$

$$t_2' = m * 2 + b$$

$$t_2' = 2 * t_0 - t_1$$

Once the expected timestep is calculated, the visibility scalar quantity r is calculated based on the difference between the expected value t_2' and the observed value t_2:

$$d = t_2 - t_2'$$

$$r = |d^p| * s + c$$

The visibility parameter has several tunable parameters p, s, and c that are adjusted to provide exaggeration. Because the value of the current timestep is known, an error function can determine how well this value was predictable. If the relative error is small, then the particle is discarded from the visualization as "uninteresting". Note that

m involves using a 2-point finite difference method to calculate slope, but the overall solution for t_2' is very similar to the temporal derivative using a 2-point finite difference. t_2' will always be within a factor of $2 + t_1/(t_0 - t_1)$ to the finite difference temporal derivative $(t_1 - t_0)$, which means when the prior timesteps t_0 and t_1 are very similar, the value for $t_2' \approx 2 * (t_1 - t_0)$. This implies that the output of this method when the difference $t_0 - t_1$ is small will be similar to simply looking at the current value of the field t_2 subtracted from the temporal derivative of the field. This does not offer a strong physical intuition which is part of the problem with this method.

As implemented, the linear history function works on the magnitude of the vector field, but could be modified to work with orientation. With good temporal resolution, this method seemed to be quite useful. However, many seismic simulations are afflicted with poor temporal resolution where there is much visual discontinuity. In these cases, it is dangerous to use linear history because it would give the wrong intuition when a portion of the field stays hidden; it may be that that portion of the field was active between timesteps, but the time slices used alias at points with similar values.

2.5.4 Orientation Difference

Although using vector magnitude with Linear History showed promise, looking at vector orientation is a key part of this work. A slightly different idea is that perhaps orientation may be more relevant as a visibility function than magnitude. To determine visibility, the orientation history function uses a slightly different approach than how orientation would be implemented with linear history.

Orientation difference only needs the prior timestep because it is not attempting extrapolation, like in Linear History, but rather visualizing the areas where difference is large. For the current timestep and the prior timestep, the vectors are converted to spherical coordinates [2]. The magnitude of the difference in spherical coordinates for the ϕ and θ components are used to emphasize particles that have drastically changed directions. This happens on the wave front and on reflective boundaries.

[2] Spherical coordinates are chosen rather than performing dot products on normalized vector direction in cartesian coordinates because it requires less memory (only 2 components are stored rather than 3 for a normalized vector).

Conversion to spherical coordinates given cartesian coordinates x, y, and z:

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = \arctan(\frac{y}{x})$$

$$\phi = \arccos(\frac{z}{r})$$

Calculating magnitude of difference given prior timestep components ϕ' and θ':

$$\mathbf{d} = \langle \phi, \theta \rangle - \langle \phi', \theta' \rangle$$

$$mag = \sqrt{\mathbf{d} \cdot \mathbf{d}}$$

Copy current coordinates for the next timestep:

$$\phi' = \phi$$

$$\theta' = \theta$$

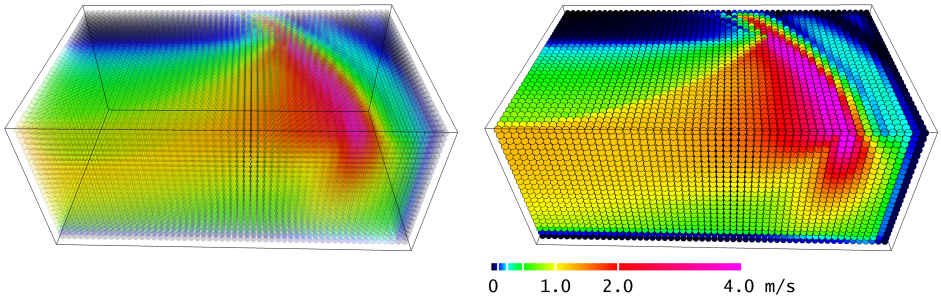
While a neat idea, it was found that the places with significant orientation differences were actually distracting from the overall patterns of wave motion. In particular, boundaries have significant orientation differences in addition to areas with mathematical noise. For instance, a small value could be fluctuating between negative and positive values, which creates significant difference with orientation. It was extremely difficult to find intuition with this method, but perhaps it may be useful with future enhancements.

2.6 Scale

Visibility functions are defined to produce one scalar value of visibility in the range 0 to 1. This value can be used by scaling glyph geometry to mask uninteresting areas. When the size of a glyph is 0, it occupies no volume and effectively transparent. The larger a glyph becomes, the more space it occupies, and the more space is occluded. With this in mind, scaling can be used as a method for displaying visibility. Glyph scaling has the added advantage that due to z-buffering, it is an order independent visibility display technique. An example of scaling glyphs can be seen in row two in Figure 2.11.

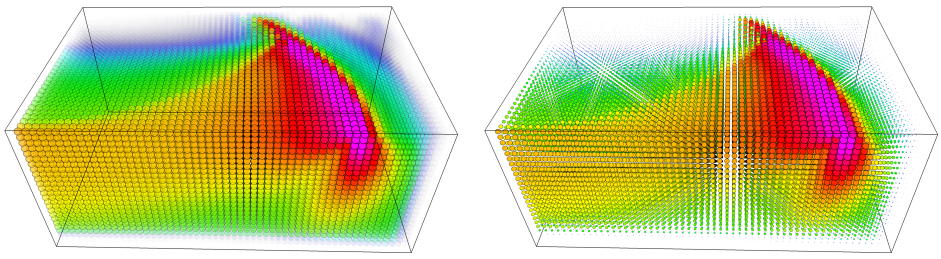
2.7 Opacity

The visibility value produced by the visibility functions can also be used with opacity. Opacity is a common method to view interior regions in a volume, frequently used by volumetric rendering systems. As opacity rendering is view dependent, it incurs an overhead and decreases interactivity. Row one in Figure 2.11 shows how opacity causes glyphs on the edge of wave fronts are difficult to discern individually even with halos. In contrast, scaled glyphs can be distinguished easily and a viable alternative that does not incur a performance setback.



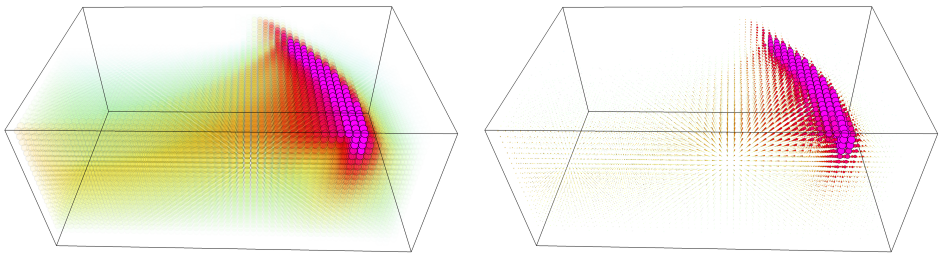
(a) Uniform Opacity

(b) Uniform Scale



(c) Logarithmic Opacity

(d) Logarithmic Scale



(e) Power Opacity

(f) Power Scale

Figure 2.11: The left column shows application of opacity function to uniformly sized glyphs and the right column shows the application of scale to glyphs. Uniform is shown in the first row, logarithmic in second row, and power (x^2) in the third row. The uniform function occludes the volume interior, while the logarithmic function helps to reveal the interior. The power scaling (x^2) is used to emphasize the kinetic energy of the wave.

Chapter 3

Context Techniques

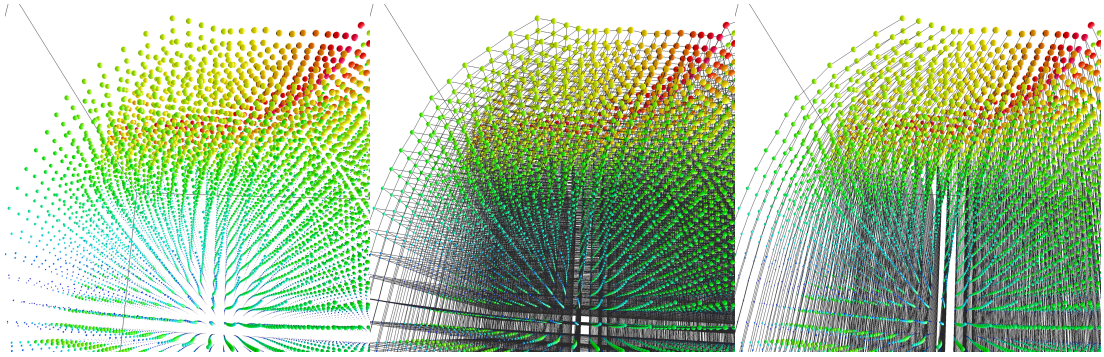
Glyphs by themselves are a useful visualization method. However, the utility of glyphs can be greatly improved by providing relevant contextual cues to the structure of the field and insight into context of the simulation of the field.

The two most prominent difficulties with glyph structure relate to distinguishing depth of glyphs and a related subject of discerning neighborhood of glyphs. If depth distinction is good enough, then this resolves glyph neighborhood ambiguities. However, no method can resolve depth to the point that one can easily distinguish a glyph's neighborhood. This motivates the method of using lattice to display and discern glyph position and neighborhood. GlyphSea employs a modern computer graphics technique of screenspace post effects to provide depth enhancing methods. Additionally, simulation data is used to provide contextual information with isosurfaces, volume slices, and maps. These methods are described in more detail in the following subsections.

3.1 Lattice

GlyphSea can intuitively visualize displacement by updating glyph position to reflect displacement at a location. However, displacing glyphs causes instances where glyph positions cannot be visually tracked with ease. This can happen when there are large visual discontinuities, such as with poor temporal resolution. In addition when the view frustum changes it is difficult to understand how glyphs have actually moved.

To alleviate visual glyph structure consistency with displacement, GlyphSea em-



(a) No Lattice (b) Full-Lattice (c) Kelp-Lattice

Figure 3.1: Panel (a) displays the glyphs with exaggerated displacement. It is often hard to discern the association of glyphs with their original location and neighborhood. To overcome this a wireframe lattice is shown in panel (b) which allows the user to see spatial associations well, however the lattice adds more clutter to already dense volume. This method is further refined to only show kelp-like vertical lines. This reduces the visual clutter while retaining spatial association.

ploys a full-lattice (Figure 3.1b) or kelp-lattice (Figure 3.1c) guides to show neighborhood context of glyphs. The full-lattice displays a grid where neighboring glyphs are interconnected by lines. This gives a strong sense of neighborhood, but it also introduces additional clutter. Inspired by undersea kelp forest we created kelp-lattice where the connecting lines are perpendicular to the ground surface, which helps to reduce clutter. Like kelp which allows the sea diver to easily view the water currents, the kelp-lattice allows the viewer to see wave propagation in volumetric seismic data.

A side effect of using lattices is that the glyph size should be small to reduce visual clutter and prevent occlusion of the lattice and volume in general. To minimize clutter GlyphSea provides an interface to customize the number of grid lines that are shown. The algorithm to provide good meshing is done with smart sampling. Simply sampling each axis with a uniform number of samples provides unnecessary clutter because most seismic volumes are not square. Instead, an algorithm was devised that samples each axis proportionally compared to the largest axis. Additionally, it is desirable to have a minimum of two sampling points per dimension that would construct a bounding box. To determine the number of samples $samples(a)$ for each axis x, y, and

z with a sampling density parameter $density$:

$$samples(axis, density) = max\left(\left\lfloor \frac{density * dim(axis)}{max(Axes)} \right\rfloor, 2\right)$$
$$axis \in Axes : Axes = \{x, y, z\}$$

where $dim(axis)$ returns the dimension of $axis$. Once the sampling amount is known, then the resolution per sample is simply computed with division creating equally sized samples:

$$resolution(axis, density) = \frac{dim(axis)}{samples(axis, density)}$$
$$axis \in Axes : Axes = \{x, y, z\}$$

3.2 Isosurface

A key interest for geophysicists is to understand wave propagation. While a velocity vector field provides useful information, it does not provide the cause of variations due to underground features like material density and stiffness. The ground characteristics of a volume used in simulation describes the stiffness of the ground that affects the wave propagation. This data can be used for computing an isosurface which identifies locations and 3D structure of the sediment filled basins (Figure 6.2). Providing this visual context is extremely important for scientists to understand wave propagation and investigation of significant basin effects and wave type conversions.

Isosurfaces are generally generated using either Marching Cubes of Marching Tetrahedra. Marching Tetrahedra is a simpler algorithm and is not patented. Prior to 2004, the Marching Cubes algorithm was patented by General Electric which motivated the initial Marching Tetrahedra algorithm. Although the algorithm requires a much more sophisticated lookup table than Marching Tetrahedra, there are robust open source implementations of the Marching Cubes algorithm which take care of these difficulties. Using an open source Marching Cubes implementation provided by Paul Bourke [Bou94] proved easier than using the Marching Tetrahedra implementations or creating an algorithm from scratch.

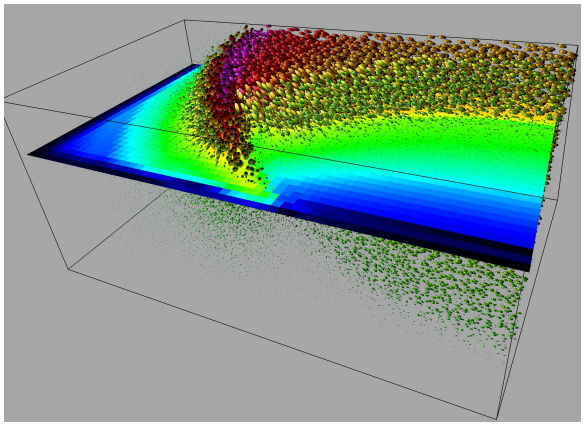


Figure 3.2: A scalar slice intersects a volume with a visibility function applied to glyphs. The slice is displaying vector magnitude with the same colormaps as the glyphs. This is useful to provide context to the areas where glyphs are not visible without providing an abundance of clutter.

3.3 Slice

Slices are a classical technique used to project a higher dimensional system to a lower dimensional system. With respect to volumetric visualizations, the interest is typically showing 2D information of a 3D volume. Moving the slices can provide an x-ray view of the volume. For the focus of GlyphSea, these slices are used to enhance rather than supplant 3D glyph visualization. The information gleamed by volume slices can provide relevant context to help understanding.

3.3.1 Scalar Slice

A textured volume slice is used to display scalar information of the volume. This is helpful for the scientist to investigate velocity magnitudes at an arbitrary volume depth. As the slice is displayed in conjunction with glyphs it can aid as a contextual cue (see Figure 3.2).

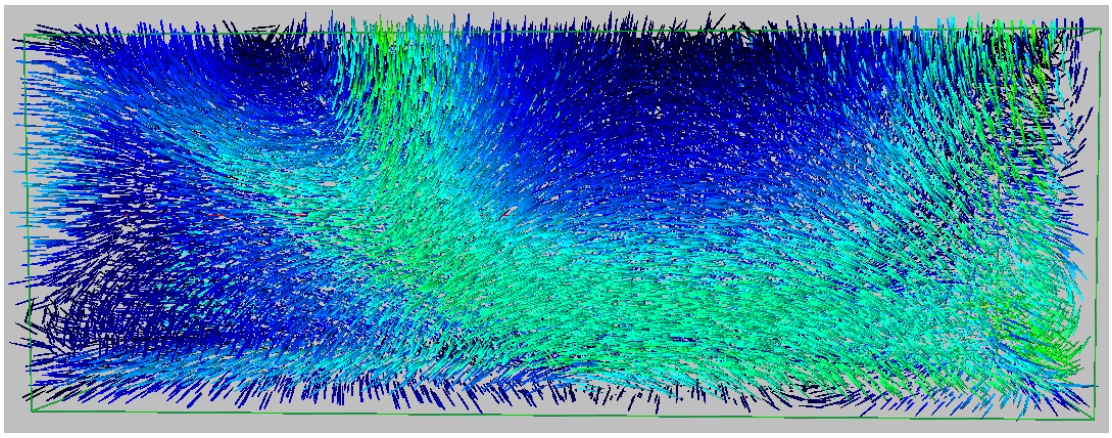
3.3.2 Fault slip rate

Ruptures along the fault are the cause of seismic waves. Initially, all wave motion is produced from this slipping of the fault. Over time and distance, reflection and refraction can impact wave motion significantly. The fault rupture dynamics are captured in a 2D slice of sliprate. This sliprate textures mapped onto the fault geometry (see Figure 6.1) provides insight into initial wave motion. Sliprate also provides a clear indication of rupture location and time when rupture stops. Display of fault rupture helps with the identification of P-waves, which are faster than S-waves. Once rupturing has stopped, the scientist can determine later arrivals associated with reflections and refraction inside the Earth. In addition, the location and physical geometry of the fault is useful for providing rich context for exploration.

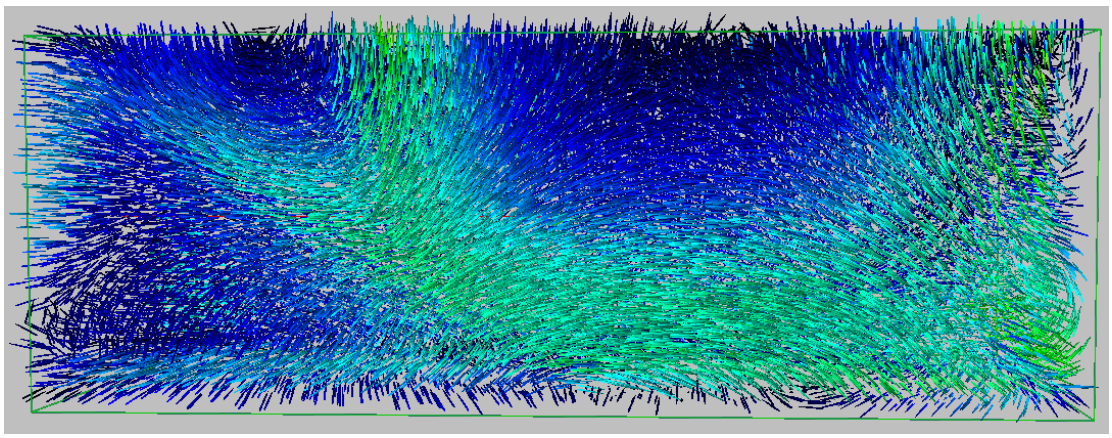
3.4 Color Jitter

Color jitter describes a technique to stochastically change the brightness of colors of the glyphs to provide a texture to discern differences in foreground and background (see Figure 3.3). The results of this method are highly dependent on the stochastic process. To provide intuitive pseudorandomness in a 4D volume the randomness needs to be unique per glyph, not change per frame, not change per timestep, not change per orientation, and not change on interactions such as camera manipulation. This disallows use of frame number, timestep, glyph orientation, and transform matrices as sources of entropy for a pseudorandom number generator because they lack visual consistency in common use situations. This leaves only a few sources of entropy to use. The source of entropy chosen was glyph position, which due to regularity of the grid, are unique per particle. The glyph position is the seed into a linear congruential pseudorandom number generator implemented on the GPU in a vertex shader.

A color jitter approximation could be created using the depth buffer, but screen space ambient occlusion is more intuitive method that supplants this technique.



(a) Without Color Jitter



(b) With Color Jitter

Figure 3.3: Color is jittered slightly based on glyph position to provide additional texture which ideally would help with glyph distinction. Unfortunately, it has the consequence of adding noise which can cause distraction.

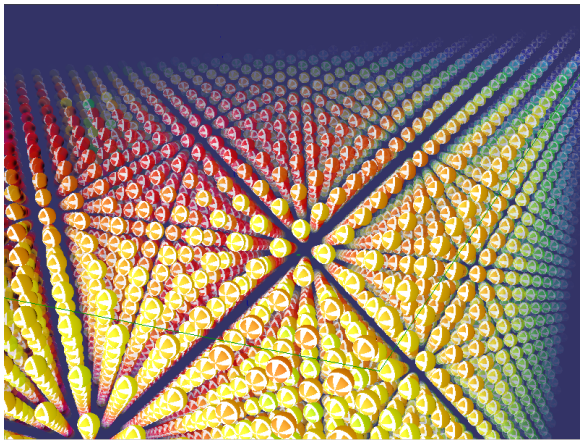


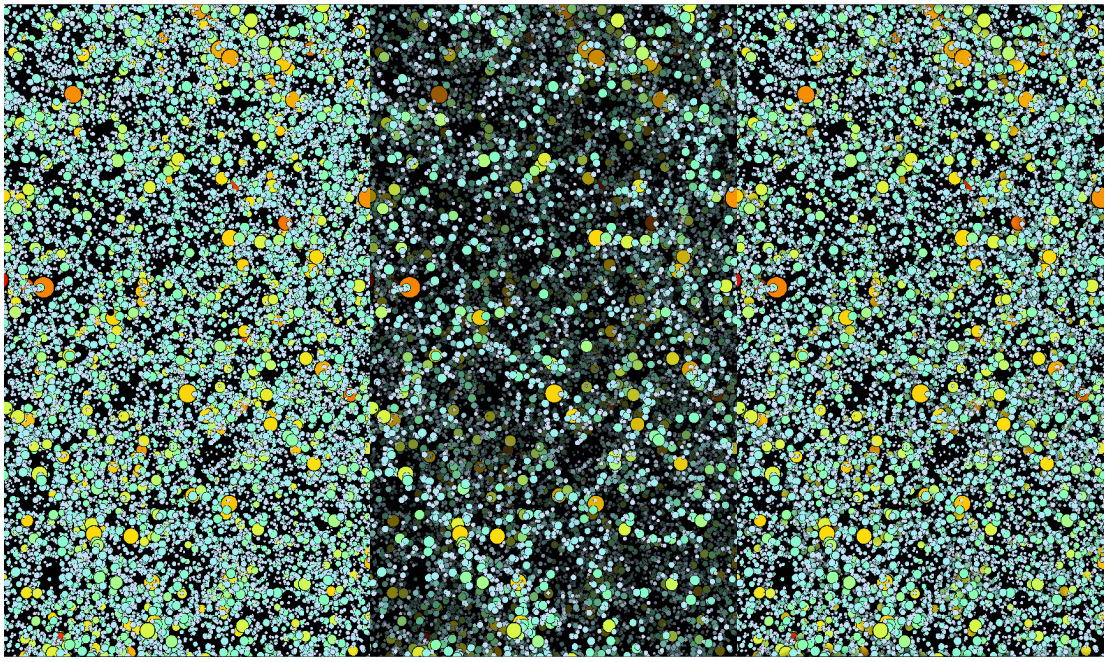
Figure 3.4: Cross mark textured glyphs are rendered without halos using Fog of Science. This smoothly enhances foreground regions but in consequence makes distant glyphs impossible to comprehend.

3.5 Fog Of Science

An early problem with glyph fields was distinguishing foreground and background glyphs. Although glyph distinction provides some cue to depth, it is not sufficient to fully resolve glyph depth. "Fog of Science" is a concept to have a transparency falloff where foreground glyphs are fully opaque and background glyphs are fully transparent, as seen in Figure 3.4. The closer glyphs are to the camera, the more opaque they become. The visual intuition comes from the illusion that glyphs are in a fog or underwater. The falloff amount is very difficult to tune properly and forces the viewers attention to camera location and not necessarily interesting data. Physical lighting depth cues proved to be more intuitive and give the viewer more freedom to look in the volume at interesting glyphs that are not necessarily in the foreground plane.

3.6 Screen Space Light Attenuation

A method very similar to transparency falloff is to change the ambient lighting based on light attenuation from the source of the camera. Light attenuation can enhance glyph depth while being intuitive as it is based on physical lighting. The mechanism implemented attaches a point light source to the camera. Because it is implemented in



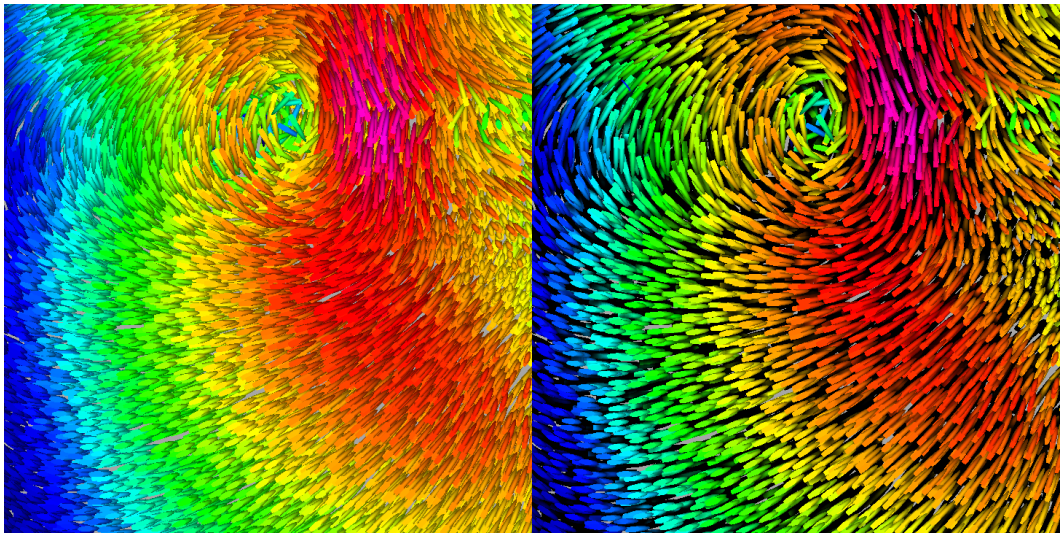
(a) SSLA off (b) SSLA on (c) SSAO on

Figure 3.5: Comparison of undirected flat shaded sphere glyph with screen space light attenuation (SSLA) or screen space ambient occlusion (SSAO) with a galaxy cluster visualization. Notice how it becomes easier to discern depth with light attenuation in sparse volumes where SSAO does not help much.

screen space, it can be flexibly enabled and disabled. The screen space implementation simply adjusts the light based on the value in the depth buffer. Although not as useful with seismic volumes due to the dense nature of seismic volumes, this technique can be quite useful when displaying galaxy clusters as seen in Figure 3.5.

3.7 Screen Space Ambient Occlusion

Ambient occlusion is able to improve depth perception and glyph distinction (see Figure 3.6). Gribble et al. [GP06] showcase real lighting models with glyphs by precomputing each glyph's texture. Instead, GlyphSea implements a screen space algorithm. The algorithm implemented unsharpens the depth buffer as described Luft et al. [LCD06]. This method was chosen because it is computationally cheap and does



(a) SSAO off (b) SSAO on

Figure 3.6: Comparison of the comet glyph with (right) and without (left) screen space ambient occlusion (SSAO).

not require normal data from the scene to be stored in the rendering pipeline; it can be added onto any existing rendering system without additional overhead or significant re-engineering. Runtime is dependent on the number of pixels rendered regardless of the scene geometry and viewing position making it beneficial for interactive systems.

However, this screen space method requires tuning several parameters. The blur kernel radius and near, far clipping planes for the depth buffer must be configured to support a field of glyphs. GlyphSea scales the blur kernel radius by depth values such that further objects have less contribution. The choice of near and far clipping planes of the depth buffer has significant impact on display thus they should be chosen carefully. OpenSceneGraph uses several automatic depth calculations specified as `osg::CullSettings::ComputeNearFarMode`:

- `COMPUTE_NEAR_FAR_USING_BOUNDING_VOLUMES`: OpenSceneGraph uses the bounding volume of scene objects to calculate near and far values appropriately. This is the default value.
- `COMPUTE_NEAR_FAR_USING_PRIMITIVES`: OpenSceneGraph uses a finer granularity than bounding volumes, instead calculating near and far planes based on

primitive (typically point, line, triangle, or quad) location.

- `DO_NOT_COMPUTE_NEAR_FAR`: The near and far planes are handled by the user and not modified by OpenSceneGraph.

Unfortunately, OpenSceneGraph's bounding volume implementation does not seem to calculate the near and far planes consistently when inside a bounding volume. Because there is one bounding volume for all glyphs, this causes the wrong behavior when exploring inside the scene. Calculating per primitive with glyphs does not work properly because glyphs are not represented as primitives in the conventional OpenSceneGraph methodology. The alternative of implementing custom near and far planes was explored, but in the end simply using bounding volumes seemed satisfactory. However, a more advanced implementation may redraw the depth buffer of the scene with a custom near and far clipping plane just for the occlusion pass. Additionally, GlyphSea exaggerates and clamps the occlusion factor to highlight objects in front (see Figure 3.6).

Typically shadowing with this method reduces luminosity, but it could also be useful for illustrative purposes to increase luminosity. This can be useful with print situations, where lots of shadows could overwhelm the colors of a glyph field. Instead of reducing luminosity, these shadows would appear to whiten rather than blacken the scene.

Screen space ambient occlusion is able to enhance depth perception and glyph distinction. It is possible to disable glyph halos with this method. It seems to work best for smaller glyphs like comets. This method is intuitive based on physical shadowing that is seen daily by users.

Chapter 4

Hardware Systems

4.1 Personal Computer

GlyphSea originally targeted desktop workstations. Development started with Linux using the Fox Toolkit for GUI and OpenSceneGraph as a high level scenegraph system for OpenGL. The initial prototypes used procedurally rendered glyphs, but were designed to do all other processing on the CPU.

After the initial Linux development, it was useful to use Visual Studio's debugger tools and this motivated the Windows development. This provided an additional feature of providing robustness to the application. Because Visual Studio's compiler produces different code than GCC, and the code links against different libraries than Linux, this helps distinguish bugs and provide robustness. In addition, the application was extended to compile on Mac OS X with minimal changes to the Linux qmake build system.

Personal computers are an ideal platform because they allow flexible dissemination of GlyphSea to scientists. The main requirements are a G80 or newer NVIDIA GPU and 2GB of memory. Although less memory could be used theoretically, using OS provided disk caching requires around 2GB with most operating systems.

GlyphSea's graphical user interface is designed to minimize clicking and mouse movements. The most common features are topmost in the GUI and promoted to the main toolbar. Sane default bookmarked values are associated with the most common widgets and are only a click away.

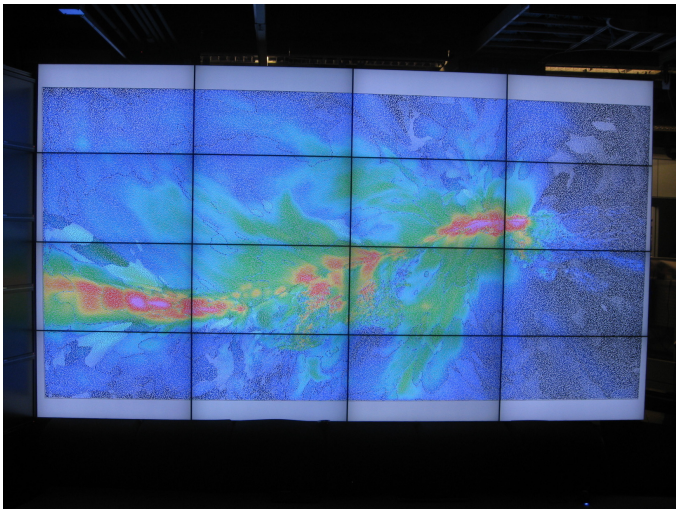


Figure 4.1: A snapshot of surface cumulative peak velocity of the Wall2Wall simulation data on a tile display consisting of 4 columns and 4 rows of 30 inch LCD monitors. Each monitor has a 1920 x 1080 pixel resolution combining to form a total effective resolution of 33 megapixels. Several distinct pockets are visible where the the velocity direction is organized in similar manner. The tile display enable us to scale the glyph geometry such that each glyph is distinct with losing the broader context.

4.2 Tile Display

A Tile Display is an arrangement of multiple display screens usually in planar tiled configuration (see Figure 4.1). The rendering engine of GlyphSea was ported to leverage CGLX with several tile systems. Both the Calit2 Hyperwall and SDSC Opti-Portal were used and provide a 34 megapixel resolution tiled display [1].

However, several things were necessary for the CGLX port. An OpenSceneGraph rendering framework was provided by Kai-Uwe Doerr. For small datasets, performance is sufficient to not require optimizations like view frustum culling. This can be performed by splitting the one large block of glyphs to be segmented into smaller blocks. It is straightforward for regular grids, but requires space partitioning schemes for irregular meshes.

A difficulty with distributed display systems is timing and randomization. All random number generators are seeded from 0 in GlyphSea, providing consistent results

[1] `http://ivl.calit2.net/wiki/index.php/Infrastructure`

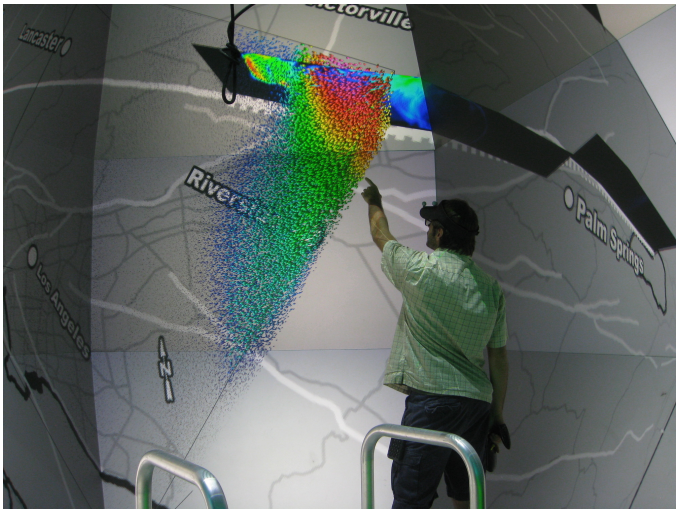


Figure 4.2: A snapshot of operation in the StarCAVE, which is a stereo capable immersive environment with an effective resolution of 34 megapixels driven by 34 HD resolution projectors.

amongst the distributed render nodes. CGLX provides a synchronized timing mechanism that is used by GlyphSea to provide consistent framerates and updates across all nodes. Otherwise, a visual "shearing" artifact is noticed where one node will render a timestep ahead of other nodes. CGLX does not provide a GUI toolkit, so currently the interface is limited to mouse movement and keyboard toggles.

In addition to technical difficulties, there were some visual difficulties. A white background can overwhelm the viewer with aggregate brightness of several screens, so a darker background is suitable for this system. The tile display [DLR+09] provides a high resolution environment which permit large number of glyphs to be displayed and perceived easily and can be used with multiple viewers

4.3 CAVE

A CAVE is an immersive display system that surrounds the user with seamless visual environment. GlyphSea uses the StarCAVE[2] [DDS+09] (see Figure 4.2) which has a 34 megapixel resolution using polarized stereo projection system with 34 HD pro-

[2] `http://ivl.calit2.net/wiki/index.php/Infrastructure`

jectors for display on a 5 wall configuration. The StarCAVE requires a different method for screen aligned billboards than the workstation and tile display implementation. This is due to the difference in coordinate transformations in the StarCAVE. The PC version has a fixed scene and mobile camera position, while the stereo system has a fixed camera and mobile scene.

Like with Tile Displays, GlyphSea use a black background to reduce overall brightness in the StarCAVE. Jittering of glyph position is essential in the stereo environment to reduce moiré patterns in 3D. GlyphSea also uses an additional per-pixel clipping plane within the fragment shader to clip glyphs too close to the viewer which causes distraction and occlusion. This is done per-pixel because the volume of glyphs appears as one scenegraph object to OpenSceneGraph, meaning that the typical view frustum calculations will not fully clip large glyphs close to the eye in a pleasing manner. For both CAVE and Tile Display it was necessary to synchronize timing to prevent shearing when transitioning between timesteps on separate rendering nodes.

The StarCAVE runs GlyphSea using COVISE with OpenCOVER. OpenCOVER provides a GUI framework that allows a fully configurable environment. The menu system is created such that submenus can be detached. Similar to the desktop, the menus were constructed in a manner that the most common functions require the least submenu depth. However, as the submenus can be detached, it allows a user to have a custom set of menus visible that show the most relevant GUI features as decided by the user.

Not only does the StarCAVE allow a flexible interface, it provides superior immersion. When a user is inside the StarCAVE, they are introduced to a seamless large resolution display where peripheral vision is filled with the visualization. The display provides stereo perspective which helps remove some depth ambiguity when glyphs are scaled, but unfortunately does not help resolve glyph shape. In addition, head tracking helps with occlusion and depth perception for a single user, allowing this user to simply move his or her head to change perspective. Unfortunately, screen space methods would need to be re-designed to work properly in a multidisplay environment, probably requiring ghost pixels to be rendered for the edges of neighboring view frustums.

Chapter 5

Implementation

5.1 Design Criteria

GlyphSea was constructed to be a tool useful for data exploration. First and foremost, it should provide intuitive methods to allow understanding of data. Secondly, the visualization should be fast enough to provide interactive performance. Because optimization was a secondary concern, there were many choices made that provide a more flexible implementation while sacrificing some performance and thrifty utilization of resources.

5.2 Visualization Engine Overview

The main loop of GlyphSea performs three phases: loading the data, computing the glyph attributes, and updating the glyphs' display. The loop is depicted in Figure 5.1, but described in detail through exploration of the class hierarchy.

The graphics engine with GlyphSea was designed around object oriented class hierarchy that provides flexibility and good performance. There are 3 base classes that most classes inherit from:

- Volume4D contains the raw volume data loaded from disk.
- Filter4D provides filters run on the raw volume data that are used for display.

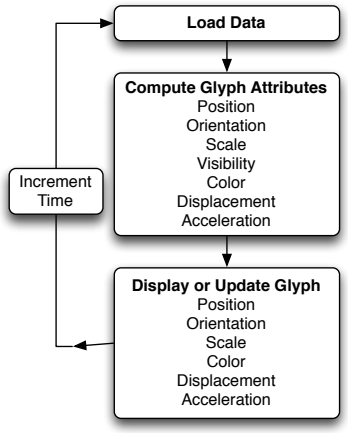


Figure 5.1: The main loop.

- DisplayVolume4D provides OpenGL or OpenSceneGraph display code for data. Data is often from a filter, but can also be based on other parameters like a map or bounding box.

Volumes for Volume4D can be of several types. All current implementations use a Component4D class that is a subclass of Volume4D. This Component4D class provides storage for a 3-component volume. Most of the glyph system is based on having a 3-component orientation vector, and as such requires a Component4D volume as input. There are three types of subclasses of Component4D: Sine, Disk, and Galaxy. Sine provides artificial data constructed from a sine wave. Disk loads time dependent volume data from a disk and expects a regular grid. Galaxy, on the other hand, reads a single frame of possibly irregular tabular data from disk. Disk is often used with seismic data, while Galaxy can support other data such as galaxy clusters.

After a volume is loaded from disk, it is passed through some set of filters that alter this loaded data before display. Some filters provide little to no modification, only connecting some components to different outputs. For instance, the OrientationF4D filter simply maps 3-components to orientation of a glyph, which can be displacement, velocity, or acceleration. The same can be said of the PositionF4D filter, except that it updates glyph position. AccelerationF4D and DisplacementF4D are filters that correspond with calculating acceleration and displacement from a Component4D velocity volume. Certain filters require indexing each glyph based on x, y, and z component

and thus are a subclass of IndexedFilter4D. It is faster for a function not dependent on indexing just to treat the component volume as one large vector, rather than a multi-dimensional array, due to the extra calculations for a multidimensional array indexing and forced memory locality of one large vector. Glyph jittering and coloring (due to interpolating between jittered positions) both require indexing and are subclass of IndexedFilter4D.

The visual display occurs with DisplayVolume4D classes. The glyph display uses the SphereD4D and SphereFilterD4D classes, while there are other display classes for heightmap, isosurfaces, lattices, bounds, textured faults, maps, and volume slices. A class called Production4D is an intermediate class that provides a layer between front end GUI or command line code and backend calls to create and remove filters, displays, screenshot, and time controls. Production4D assumes that a glyph system is being rendered and constructs the appropriate backend features. All front end implementations (FOX, CGLX, COVISE, command line) utilize Production4D to increase code re-use and reduce complexity for frontend implementations.

GlyphSea is implemented in 33260 lines of C++, Cg, and GLSL. It uses 4124 lines of code for the Fox Toolkit GUI, 829 lines of code for the CGLX interface, and 1372 lines for the OpenCOVER interface. Cg shaders for glyphs comprise 5470 lines of code, while GLSL shaders used in the post effects pipeline comprise 226 lines of code. A source file count can be seen in Table 5.1 that gives additional insight.

GlyphSea employs several support libraries. OpenSceneGraph[1] provides a scene graph engine on top of OpenGL. This has enabled porting GlyphSea to all major operating systems (Linux, Mac, and Windows) and high resolution display systems. For workstations, GlyphSea uses the Fox Toolkit for display. For tile displays, GlyphSea uses CGLX[2] to provide a distributed OpenGL rendering system. For the StarCAVE, GlyphSea uses COVISE with OpenCOVER [3] for distributed rendering and GUI support.

GlyphSea uses billboarded glyphs rendered with Cg shaders. Billboard orientation is handled within the vertex shader, while shading techniques are applied in the

[1] `http://www.openscenegraph.org/`
[2] `http://vis.ucsd.edu/~cglx/`
[3] `http://www.visenso.de`

Table 5.1: Count of source files in the GlyphSea source directories. The number of files gives an estimate of complexity for the various systems. Notice that the shaders comprise a nontrivial portion of source code. This is not ideal, as if a subtle bug is found in one shader, then this likely means many will need to be modified. However, it is difficult to create a system that is both efficient and has good code reuse.

GUI and Platform Support Source Files		
Folder	Description	Count
foxe	Custom FOX Toolkit GUI widgets	44
opencover	OpenCOVER GUI and support files	27
vize_fox	GlyphSea window implementation	5
cglx	CGLX support files	4
windows	Windows specific support files	2
linux	Linux specific support files	1
Total		84

Core Engine Source Files		
Folder	Description	File Count
vize	Graphics engine for GlyphSea	108
osge	Custom OpenSceneGraph extensions	47
shaders	Cg and GLSL shaders for glyph rendering and post effects	44
filters	Volume filters for functions like marching cubes, curl, divergence, and gradient	23
type	Global variable registry and math support functions	16
Total		238

fragment shader. Ellipsoids are drawn with a technique similar to that described by Gumhold in [Gum03].

Glyph positions are updated within the CPU to simplify implementation, debugging, and prototyping. As an optimization, GlyphSea performs state checks to reduce copies to and from the device. This ensures that data is only copied to the GPU when values have changed. Additionally, code is written to be friendly to the CPU prefetcher by reducing loop complexity. Glyph state is not updated with the traditional OpenSceneGraph state operations, but rather modified via direct array modifications. These arrays are then copied to their respective OpenGL arrays. This is similar to how animated textures are implemented with OpenSceneGraph. Future efforts may reduce the copies by performing adjustments within the GPU using überbuffers [KSW04][KLRS04], CUDA, or OpenCL.

GlyphSea uses a novel application of screen space ambient occlusion within an interactive scientific visualization context. In addition, GlyphSea implemented screen space light attenuation and depth of field blurring, but these are less useful for seismic fields. All screen space methods are implemented within a post effects pipeline. A post effects pipeline can be described in several steps. First, the scene is rendered to a camera with an offscreen framebuffer. Post effects methods are applied to this offscreen framebuffer. This framebuffer is then mapped to a screen aligned quad which has the merged results of the post effects. This screen aligned quad is essentially a billboard that fills the screen. In stereo environment, the depth buffer must be updated in this screen aligned quad or else the depth would appear flat.

5.2.1 Theoretical Memory Usage

A theoretical bound for memory usage is constructed in the following section based on code analysis. Although all arrays are constructed at some point, unused arrays are deallocated. In addition, structures may be padded to align to various memory boundaries, so this analysis presents a minimal worst case description of memory usage.

Reducing memory usage would affect performance in several beneficial ways, mostly by reducing memory transfers within the CPU and memory transfers from the CPU to GPU. A significant cost is storing the arrays necessary for glyph state within

the CPU. With quad billboards and vertex lists, the current solution allocates arrays for coordinates, radius, orientation, and colors that cost 208 bytes per glyph. The glyph data that is modified by various filters has several fields that sum to 80 bytes per glyph: radius, 3-component position, 3-component origin, 3-component direction, 4-component color, 3-component acceleration, and 3-component displacement. Because the current implementation streams data from disk, the whole dataset is not explicitly loaded into memory but rather only one timestep. Therefore the total cost for loading the velocity data is 12 bytes per glyph for the x, y, and z components. However, modern operating systems will likely cache the data from disk into memory, such that performance is better after playing once through a dataset. Other sources of memory include the filters themselves. The default filters include acceleration, displacement, linear history, and orientation history. The combination of all of these filters allocate 48 bytes per glyph. In total, each glyph requires 348 bytes with the current implementation as seen in Table 5.2. Some of these bytes are redundant, but others are necessary. A smart solution would chain operators together for filters rather than copying memory around. Fortunately, most engine features are only enabled selectively, causing the memory access overhead per glyph to be less than the worst case 348 bytes.

Due to the closed nature of GPU hardware, it is difficult to ascertain exact memory usage. However, a reasoned approximation can be made based on data required in the vertex and pixel shaders in the rendering pipeline. Each vertex requires four component vectors: normal, color, position, and texture coordinates. The “normal” actually encapsulates the non-normalized field vector, which is normalized in the vector shader. The color is four components to support alpha values. Position is also four components because of homogeneous coordinates. The texture coordinates are used to distinguish different corners of a billboard and also to store the radius in the “z” component. These four floating point vectors require 64 bytes of memory per vertex. There are four vertices per glyph, so the total storage cost per glyph is 256 bytes as seen in Table 5.2.

Using pointsprites could reduce storage to 64 bytes per glyph. It may be possible to pack floating point values of the vectors and further save memory, but this requires losing precision which is undesirable in a scientific setting. However, if temporal resolution is sufficient that there are minor differences between timesteps, then a keyframing

Table 5.2: Minimum bounds for memory usage for various common volume sizes along with GPU Bandwidth necessary to sustain 30 frames per second. These are theoretical minimums found by looking at memory allocation within the code. Additional features, such as memory padding, isosurfaces, maps, and textures may affect actual size. With current motherboards (generally those built after 2008), the limiting factor for GPU bandwidth is the PCIe 2.x interconnect that provides 8GB/s of data throughput. It is much more common for systems to have PCIe 1.x, which provides a 4GB/s interconnect. If the data only resided on the GPU, then the bottleneck would be global memory bandwidth on the GPU with 104 GB/s for a high end Quadro 5800FX. This Quadro has 4GB of memory, so memory bandwidth would become a concern before occupying the full memory with the current implementation.

Glyph Count	CPU Memory	GPU Memory	GPU Bandwidth 30hz
10^3	0.33 MB	0.06 MB	0.002 GB/s
100000	33.19 MB	24.41 MB	0.715 GB/s
100x125x20 *(TS2.1 crop)*	82.97 MB	61.04 MB	1.788 GB/s
64^3	87.00 MB	64.00 MB	1.875 GB/s
1000000	331.88 MB	244.14 MB	7.153 GB/s
128^3	696.00 MB	512.00 MB	15.000 GB/s
256^3	5568.00 MB	4096.00 MB	120.00 GB/s
750x375x100 *(TS2.1 full)*	9334.09 MB	6866.46 MB	201.166 GB/s

solution may be useful where a keyframe sends the full valued float, and then between keyframes packed floating point values can be used. This is a data compression mechanism that has definite guarantees on precision and accuracy, but provides no guarantees of compression quality when the temporal resolution is poor and the differences between timesteps cannot be quantized to a certain error threshold.

Graphics runtime for the number of glyphs heavily depends on the size of glyphs because the pixel shader has more pixels to render with a large glyph. Runtime is $O(n * r^2)$, where n is the number of glyphs and r is the glyph radius in pixels which depends on depth from the camera. With large volumes having large values of n, large radius glyphs can quickly bring the system to a halt. This motivated a clamping mechanism that clamps the radius to less than 50 units.

5.3 Geometry Implementation

There are two significant rendering methods commonly used today: rasterization and raytracing. Raytracing projects rays from a camera through pixels into the scene and returns color value for each pixel. On the other hand, rasterization projects the scene onto pixels in imagespace. The traditional rendering pipeline uses rasterization with triangles as the base primitive for scene. This can cause issues with scene complexity for rendering smooth geometries like spheres, ellipsoids, cones, cylinders, saddles, or donuts. To compensate for this, many solutions employ "billboards", which are squares aligned with the camera that project a "splat" texture of the object with the proper perspective.

5.3.1 Billboard

The billboarding technique can be described as follows. Four points compose a `GL_QUAD` for each glyph. All four points have the same position given as input to the glyph rendering class called "coSphere". These points are defined either in a vertex buffer object or with traditional `glBegin() glEnd()` statements. Vertex buffer objects are faster than `glBegin() glEnd()` statements but require additional complexity to use arrays with certain culling systems such as contribution culling. On the other hand, it

is trivial to implement culling mechanisms with the `glBegin() glEnd()` statements. The code currently supports both via preprocessor definitions, but only performs contribution culling without vertex buffer objects. Geometry is sent to the vertex shader in the rendering pipeline where the points are projected to camera space. Before this transformation occurs, the billboard is created by creating a billboard aligned to screen coordinates.

Drawing geometry requires either rendering geometry to a texture in realtime, raycasting the texture splat, or prerendering the texture. Prerendering the texture has a disadvantage of being limited with respect to perspective and scaling, while rendering geometry is still expensive. GlyphSea uses a solution based on raycasting the texture splat for its ease in implementation and accuracy in per-pixel approximation of glyph geometry. Raycasting a texture splat has a disadvantage that it requires more calculations than a texture lookup. However, with increasing computational GPU performance, this disadvantage is diminished to the point that it is reasonable to calculate cheap intersection tests for realtime performance.

The current desktop vertex shader implementation constructs a screen aligned billboard as follows. Initially all four quad coordinates are set to the same position within the CPU. However, the texture coordinates are unique per vertex. The top left corner is specified as $\langle -1, 1 \rangle$ and the bottom right corner $\langle 1, -1 \rangle$. The third texture coordinate contains the "radius" or size of the billboard. The vertex shader uses these texture coordinates to move the vertex positions to create a billboard. The position offset $\mathbf{p_o}$ in model view coordinates can be calculated by multiplying the first two components of the texture coordinate $\mathbf{t_{xy}}$ by the radius. The output position $\mathbf{p_{out}}$ is created from the incoming position $\mathbf{p_{in}}$ with the model view matrix M and projection matrix P:

$$\mathbf{p_o} = \mathbf{t_{xy}} * r$$

$$\mathbf{p}' = \mathbf{p_o} + \mathrm{M}\mathbf{p_{in}}$$

$$\mathbf{p_{out}} = \mathrm{P}\mathbf{p}'$$

Finally, the orientation vector of the glyph transformed by the model view matrix, which is used with ellipsoid geometry and orientation enhancing pixel shaders.

There are three main intersection tests used: sphere, ellipsoid, and comet. The comet intersection test is a modification of the ellipsoid test, which in turn is a modification of a sphere intersection test. This leads to a natural order of describing the intersection tests.

5.3.2 Sphere

Spheres require a very simple intersection test. Because the billboard is aligned with the camera, the texture coordinates are linearly related to imagespace coordinates. Texture coordinates are typically specified in the range $[0,1]$ but are useful with glyph rendering to have an origin in the center of the texture such that the new range is $[-1,1]$. This is done on the CPU. Given texture coordinates $\mathbf{p}$, a sphere with radius r (in terms of texture coordinates), and a sphere with center $\mathbf{c} = \langle 0,0 \rangle$, the intersection test is the following:

$$||\mathbf{c} - \mathbf{p}|| \leq r$$

This condition holds where the imagespace location is within the sphere volume. Intersections tests for the sphere, ellipsoid, and comets happen within the pixel shader and are fairly cheap. When the intersection test fails, the pixel is discarded and another glyph must be drawn.

Once the imagespace aligned x-y coordinates are known for a sphere, then the depth of intersection can be found by solving for z:

$$z_o = \sqrt{r - x^2 - y^2}$$

This z_o is actually the offset from the billboard, so that the actual z_v in model view coordinates would be $z_v = z_o + billboard_z$. As long as glyphs do not intersect, it is sufficient to simply use the depth buffer calculations for the billboard rather than offsetting the billboard coordinates. When spheres intersect, proper intersection needs appropriate depth calculations. Calculating depth buffer position is done within the fragment shaders as follows. The offset model view coordinate z_o is used to create a vector $\mathbf{p} = \langle 0,0,z_0, billboard.w \rangle$. This vector is transformed by the model view projection

matrix MVP:

$$\mathbf{p}' = \text{MVP}\mathbf{p}$$

Depth is calculated by performing the homogenous division by w, producing the interval $[-1,1]$:

$$z = \frac{\mathbf{p}'_z}{\mathbf{p}'_w}$$

Output to the depth buffer z_d needs to be in the interval $[0,1]$ for OpenGL:

$$z_d = \frac{z+1}{2}$$

Finally, z_d must be set to the depth buffer output within the fragment shader. This operation is done with assigning `OUT.depth` in Cg or `gl_FragDepth` in GLSL.

Surface normals are necessary in order to provide useful texturing and shading of surfaces. For a sphere, the normals can be calculated very simply. If a ray intersects with surface location $\mathbf{p}$ and the sphere has origin $\mathbf{o}$, then the normal $\mathbf{n}$ is:

$$\mathbf{n} = \frac{\mathbf{p}-\mathbf{o}}{||\mathbf{p}-\mathbf{o}||}$$

In model view coordinates, the sphere is centered around the origin $\mathbf{o} = \langle 0,0,0 \rangle$, resulting in the simplification:

$$\mathbf{n} = \frac{\mathbf{p}}{||\mathbf{p}||}$$

5.3.3 Ellipsoid

Ellipsoids use a more sophisticated method based on work done by Gumhold [Gum03]. The intersection test for an axis-aligned ellipsoid centered around the origin is typically described with the following equation:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 \leq 1$$

Where r_x, r_y, and r_z are the radius for the principal axes of the ellipsoid. With a slight modification, the equation can be simplified for the case when it is desired to uniformly increase the size of the ellipsoid:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 \leq r^2$$

The intersection test can be further simplified due to the scene is structured. If the ellipsoid is placed axis-aligned and centered about the origin, then camera rays will all be $\mathbf{v} = \langle 0, 0, 1 \rangle$. This means that to test for intersection, the two relevant components are x and y, such that one can effectively set z to zero for the case of detecting intersection. This equation is cheap to compute, can be done efficiently with vector hardware.

Although this method is convenient and cheap, the difficulty is that ellipsoids in this application are not axis aligned. Part of the difficulty comes due to oriented geometry. Because all perspectives of a sphere produce the same appearance, a very simple view-independent intersection test can be used. This is not the case for ellipsoids. A cheap method based on [Gum03], which is described below.

Gumhold [Gum03] noticed that testing the intersection of a sphere is easier than an ellipsoid. He constructed a method that maps points to from an ellipsoid to a sphere. Some rotation R provides an orthonormal basis for the ellipsoid axes which may not align with the camera's axes. This is constructed by a slightly different method than described in [Gum03]. GlyphSea constructs a rotation matrix using the concept that in the current implementation the axes have fixed lengths and the principal axis corresponds with the x-axis **x** before projection. After projection, this principal axis needs to be oriented with **d**. The rotation matrix is then defined by an axis **a** and an angle θ that rotates **x** to **d**:

$$\mathbf{a} = \frac{\mathbf{x} \times \mathbf{d}}{||\mathbf{x} \times \mathbf{d}||}$$

$$\theta = \arccos\left(\frac{\mathbf{x} \cdot \mathbf{d}}{||\mathbf{x}||||\mathbf{d}||}\right)$$

This method works because the non-major axes of the ellipsoid have equal length and thus have rotational symmetry about the major axes. This would not be ideal if a texture or glyph were non-symmetrical. This rotation matrix $\mathrm{R}(\mathbf{a}, \theta)$ is constructed within the vertex shader.

An ellipsoid is considered "axis aligned" when the principal axes project to the x, y, and z axes. The math for calculating ray-ellipsoid intersection is simpler when an

ellipsoid is axis aligned. Within the pixel shader, the texture coordinate positions $\mathbf{t}$ are rotated to produce an axis aligned ellipsoid, mapped from an axis aligned ellipsoid to an axis aligned sphere, and then inverse the rotation to provide a non-axis aligned sphere test [4]. This is performed with the following steps where the ellipsoid axes are of lengths $\langle r_x, r_y, r_z \rangle$:

$$\mathrm{S} = \begin{bmatrix} 1/r_x & 0 & 0 \\ 0 & 1/r_y & 0 \\ 0 & 0 & 1/r_z \end{bmatrix}$$

$$\mathbf{t}' = \mathrm{R}^{\mathrm{T}}\mathrm{SR}\mathbf{t}$$

Testing for intersection is now simply testing for sphere intersection. This is accomplished with the familiar test ${\mathbf{t}'_x}^2 + {\mathbf{t}'_y}^2 \leq r^2$ that looks for the intersection point of a sphere with the plane $z = 0$. Once xy intersection has been found, project this location onto the sphere to solve for depth:

$$\sqrt{r^2 - (x^2 + y^2)} = z$$

Solving for depth requires the following 3 GPU assembly instructions [5]:

```
DP2R  R0.x, R0.wxzw, R0.wxzw;
SGTRC HC.x, R0, c[1];
KIL   NE.x;
```

Because this solves for depth of the axis aligned ellipsoid centered around the origin, the final depth is calculated by adding this extra depth to the billboard z-depth as described with spheres.

5.3.4 Comet

Comets are actually very thin ellipsoids with a cap removed. Removing the cap is performed by testing the length along the principal axis, and clipping a certain radius

[4] Because the ellipsoid is splatted on an axis-aligned billboard, texture space and screen space coordinates are constructed to be identical.

[5] The compiler performs optimizations that cause unusual swizziling on the vector `R0` to produce the `R0.wxzw` operand.

away from the origin of the ellipsoid. This is done by rotating the principal axis so that it is axis aligned, and then measuring the distance along this line. If the distance is greater than 0, it is on one half of the ellipsoid. Then a refinement step determines if the distance to the texture coordinate is more than a small radius to create a round head.

Using the parameters described in the ellipsoid section, distance along the principal axis is calculated as $d = (\mathbf{Rt})_x$. This can be stored in the fragment shader with the original ray intersection tests rather than requiring an additional matrix-vector multiplication. If $d > 0$, this is the positive side of the ellipsoid's principal axis (the direction it is pointing). When this is the case, the pixel shader determines if a glyph is visible if $\mathbf{t}_x^2 + \mathbf{t}_y^2 > 0.155$. The value 0.155 was determined through observation of what seemed to be a good tip length.

5.3.5 Twigs

Orientation would give the direction of the vector, while line length represents the magnitude. While a naïve solution would center the line, this would remove the sign of the vector. If the line is always drawn with a start point p_0 at an unchanging origin, and continues for length $l = ||\mathbf{v}||$ in the vector normalized direction $\mathbf{v}' = \mathbf{v}/||\mathbf{v}||$, then this glyph can represent direction and magnitude sufficiently. The end point p_1 can be simply calculated based on some scale s:

$$\mathbf{p_1} = \mathbf{p_0} + s * l * \mathbf{v}'$$

While l will always be positive, $\mathbf{v}'$ preserves the sign of $\mathbf{v}$. This equation reduces to:

$$\mathbf{p_1} = \mathbf{p_0} + s * \mathbf{v}$$

GlyphSea uses these equations to create `GL_LINES` primitives.

5.4 Texturing Implementation

5.4.1 Dipole Texturing

Given the vector direction **v** and surface normal **n**, dipole texturing is calculated for two different colors for the antipoles of the geometry. Both poles share similar lighting equations, with a diffuse-like component I_d, specular-like component I_s, and dipole attenuation factor s:

$$I_d = \mathbf{v} \cdot \mathbf{n}$$

$$I_s = {I_d}^s$$

Diffuse-like intensity I_d provides a subtle effect that enhances the exaggerated specular-like intensity I_s. A pole test is created to determine which pole is being rendered by looking at the sign of the dot product of the vector direction and surface normal:

$$sign = \mathbf{v} \cdot \mathbf{n}$$

If *sign* is less than zero, then the pixels being rendered are on the "dark" side of the glyph. When this is the case, then both I_d and I_s are set to negative values. Due to color clamping, a negative value will bring the color to $rgb = \langle 0,0,0 \rangle$ which represents black for the dark side of the glyph. The final color is a blending of the dipole color and the ambient color:

$$color_{out} = K_a * color_{in} + K_d * I_d + K_s * I_s$$

where typically $K_a + K_s > 1$ to provide strong dipole distinction. Current parameters were selected by trial and error to determine a good blending with the values depending on glyph geometry. For the ellipsoid glyph, $s = 100$, $K_a = 0.7$, $K_d = 0.4$, and $K_s = 0.6$. The sphere glyph has slightly different values: $s = 10$, $K_a = 0.8$, $K_d = 0.3$, and $K_s = 0.8$.

5.4.2 Cross Texturing

Cross texturing is the only non-symmetrical texture that has been experimented with. The final iteration used four triangles that meet at a point where the vector orientation is. This can be procedurally generated using the following technique.

If the vector **v** represents orientation of the glyph, then the two vectors **y** and **z** that create an orthonormal basis with **v** create a plane on which the normal **n** is projected. If the projected normal is within an angle tolerance θ_ε to the orthonormal basis axes **y** and **z** in the projected plane, then the color strip is drawn. This is performed with the following operations:

$$test_y = \frac{\arccos(\mathbf{pos}_{yz} \cdot \mathbf{y}_{yz})}{||\mathbf{pos}_{yz}||}$$

$$test_z = \frac{\arccos(\mathbf{pos}_{yz} \cdot \mathbf{z}_{yz})}{||\mathbf{pos}_{yz}||}$$

The triangle color is drawn over the current shading if $test_y < \theta_\varepsilon$ or $test_z < \theta_\varepsilon$. A value for θ_ε was empirically determined to be pleasing with $\theta_\varepsilon = \pi * 0.08$. Initially this triangle color was pure white, but this was found to quickly over brighten images and hurt comprehension. Refinement changed the triangle color to a blended function. Where **c** is the flat color determined with a colormap for the vector quantity, triangle color is defined to be $\text{clamp}(\mathbf{c} * 0.45 + \langle 1,1,1 \rangle * 0.65, 0, 1)$.

5.4.3 Concentric Ring Texturing

Concentric ring texturing uses three rings to display orientation. Each ring is created surrounding the major axis. Narrow bands are drawn along this axis by simply using conditionals to only allow intervals along the principal axis. This is performed with a smart comparison to the orientation vector **v**, which is also the major axis. Given an interval $[a,b]$ that defines a concentric ring's position and width, if $\mathbf{v_x} > a$ and $\mathbf{v}_x < b$ then a solid color is output that defines the concentric ring. The current implementation has 2 rings and one cap with the intervals specified as $\{[-0.05, 0.25], [0.63, 0.75], [0.97, \infty]\}$.

5.5 Post Effects Implementation

Post effects are a new method of realtime graphics enhancement allowed by the programmable graphics pipeline. Post effects are rendered after the scene, using the render buffers created during normal rasterization. It then post-processes these buffers and composites a final image which is used for display.

Because the rendering system was not created with deferred shading in mind, the only output of rendering available to the post effects pipeline are the depth buffer and color buffer. A deferred shading solution would require a drastic re-implementation, and would probably be easier to do within OpenGL than OpenSceneGraph because OpenSceneGraph's camera-buffer attachment system is not fully developed for a robust deferred shading solution. However, just using the color and depth buffer can provide interesting and useful effects.

The advantage of using post effects is delaying computation until the scene is rendered in imagespace. Given some section of code with cost c_p, and a viewport with width w and height h, then runtime will be $O(w*h*c)$ for the post effect. If the function can be similarly implemented with cost c_o within the pixel shader of a scene object, then this runtime would be $O(n*r^2*c_o)$, where n is the number of glyphs and r is the radius of the glyph in pixels. This leads to the realization that a function should be implemented in screen space when $O(w*h*c_p) < O(n*r^2*c_o)$.

Take for instance the motivation for performing screen space light attenuation, where $c_p = c_o$. For a reasonable scenario where the viewport is HD ($w = 1920$, $h = 1080$) and glyphs occupy $r = 20$ pixels, then it would be advantageous to use a post effect implementation when there are more than 5184 glyphs. This is almost always the case with seismic volumes.

This cost metric does not take into account bandwidth considerations, which may be significant with large imagesizes or volumes. Again, the tradeoff is the same: bandwidth of moving imagespace pixels versus the bandwidth of moving glyphs with their associated data.

5.5.1 Screen Space Ambient Occlusion

First, the scene is rendered to a camera with an offscreen framebuffer. This framebuffer is then mapped to a screen aligned quad which runs a blur pass of the depth buffer, and then an occlusion pass contributes shadowing to the scene. The final pass in the post effect pipeline renders a texture of the offscreen colorbuffer.

There are several configurable parameters useful with visualization. The occlusion coloration can be exaggerated to provide more contrast. Without the contrast the effect is more subtle. A simple way to exaggerate is to scale the depth difference Δ by some value s and clamp to $[0,1]$. While this provides a linear shift, for contrast it is more ideal to have small effects minimized while larger effects exaggerated. This can be done simply by squaring the value $(\Delta * s)^2$ and clamping to $[0,1]$. Typically shadowing with this method reduces luminosity, but it could also be useful for illustrative purposes to increase luminosity. This can be helpful for print situations where darkening an image can waste ink.

Another parameter is the blur kernel radius. Increased blur kernel radius means the depth buffer value can effect values farther from the center of the blur kernel; in other words, a larger blur kernel radius allows for a glyph to effect other glyphs with decreased spatial locality. For increased performance, it is important to reduce the number of texture fetches used in the pixel shader, which correlates with the blur kernel radius. Typically this is done by performing the blur on a down-sampled depth buffer, but unfortunately this was problematic within the OpenSceneGraph implementation. Instead, the operations run on a full resolution buffer and skip pixels. This can cause sampling artifacts, so a Gaussian noise texture is used to slightly offset sample location by as much as one depth buffer texel. Subpixel locations will be linearly interpolated in hardware, so edges become smoother in practice.

It is also important to consider the near and far plane of the depth buffer; this has drastic effects on perceived occlusion with this method. It can be desirable to redraw the depth buffer of the scene with a custom near and far clipping plane just for the occlusion pass. Additionally, GlyphSea exaggerates and clamps the occlusion factor to highlight objects in front (see Figure 3.6).

With a screen space ambient occlusion method, the results are similar to depth

dependent halos described by Everts et al. [EBRI09] without the cost of additional geometry. In addition, runtime is dependent on the number of pixels rendered, not the number of glyphs as the case with creating additional geometry. This is important with glyph systems where there is a lot of geometry rendered. Because screen space ambient occlusion occurs in screen space, it has view-independent performance that is beneficial in interactive systems.

In comparison with offline ambient occlusion, screen space ambient occlusion does not require pre-computed luminance textures and the overhead required for animated glyph scenes as described by Gribble et al. [GP06]. This provides a simpler implementation while approximating the relevant features of ambient occlusion. The main drawback to screen space ambient occlusion is that the near and far clipping planes must be appropriately selected to provide a good distribution of values in the depth buffer. This can be difficult for sparse datasets, but a Bavoil et al. [BS09] describe a depth peeling solution that can be used to increase quality.

5.6 Transparency... or lack thereof

Transparency can be difficult with rasterization. Because graphics cards use a single z-buffer to implement the painter's algorithm, only the closest z-value is guaranteed to be drawn properly. This means that objects that are behind a transparent object may or may not be drawn in the proper order.

One method to implement proper transparency is to have the objects ordered by depth such that the furthers objects are drawn first. This allows for transparent blending in the graphics card and proper transparency will be achieved. Unfortunately, reordering geometry by depth can be very costly, especially with hundreds of thousands of glyphs. This is what motivates space partitioning schemes which can reorder scenes very quickly based on camera location. With a uniformly distributed grid, the ordering of glyphs is implicit and can be leveraged for proper transparency. Without any modification, there exists one view quadrant that draws glyphs in the proper order for transparency, which is what was used for testing. Transparency was not compelling enough to motivate the effort for a full implementation where transparency is correct from all quadrants.

An active field of research is to have order independent transparency, such that scene geometry does not need to be reordered to provide proper transparency. Some promising techniques are based on depth peeling, where multiple z-buffers are calculated for each scene. These depth buffers can then provide proper rendering order for as many peels of the depth buffer that are computed. Order independent transparencies main advantage is that all computation happens within the GPU. This is important to reduce transfers between the CPU and GPU that would be necessary if the scene is reordered using space partitioning or other methods.

In addition do difficulties in implementation, transparency would cause issues with the post effects system. Because post effects are only computed in imagespace, it disregards transparently occluded glyphs. This causes visual inconsistencies particularly with ambient occlusion techniques. Unfortunately, for complex effects like screen space ambient occlusion, rendering properly would disregard the largest advantage in screen space algorithms: due to calculating per viewport pixel, rather than per glyph.

Chapter 6

Results and Discussion

GlyphSea was employed to visualize two different sets of seismic data. The exploration of the datasets was carried out with the help of Dr. Jean-Bernard Minster, Dr. Steve Day, Dr. Kim Olsen, and Dr. Geoffrey Ely several times during the course of application development and their input was incorporated iteratively. In each exploration session the application was driven by a visualization expert and seismologists provided the guidance on where and what to explore by refining entire array of parameters including geometry, texture, scale, displacement, lattice, and context cues like slice and isosurface. The system is capable to display the temporal data at a desired rate, allowing the scientist to use standard play, pause, forward, and reverse controls to move to a desired time step. Most of the exploration sessions were conducted on a PC workstation and few were conducted in immersive StarCAVE environment. A description of the two use cases is provided in the following sections.

6.1 Synthetic Data Use Case

Dr. Geoffrey Ely provided a base case simulation that consists of a square plane fault rupture. This simulation is simple and the wave propagation behavior well understood by geophysicists. The data is in the form of a uniform grid of 56 x 34 x 24 voxels with 750 time steps at 0.16 second time intervals. The data provided consists of three fields for velocity along the x, y, and z axes. This synthetic data provided a valuable testbed for exploring techniques. Although interactivity was initially used to promote

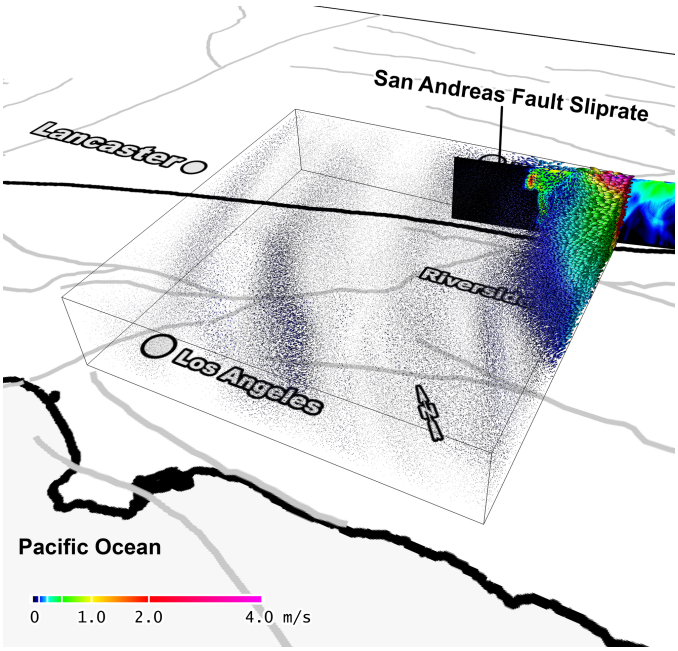


Figure 6.1: Shows the TeraShake velocity data with sliprate displayed along the vertical plane on top right, and a context base map at the bottom. Notice the P-waves preceding slip along the fault, as P-waves move faster than the fault rupture.

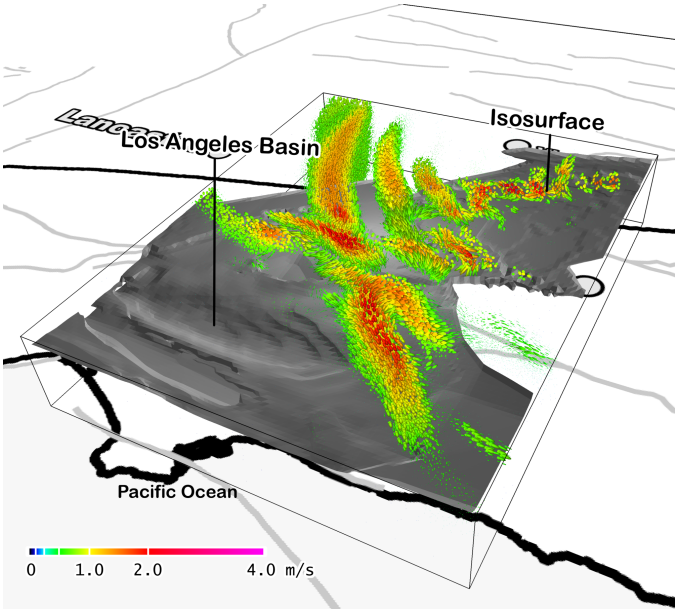


Figure 6.2: The TeraShake volumetric velocity data is drawn with comet glyphs combined with an isosurface showing basin to provide context for wave propagation in the Los Angeles basin.

rapid prototyping and adjustments, it quickly became clear that interactivity is extremely useful when exploring volumetric data. This volume is depicted in Figure 3.4, Figure 3.3, Figure 3.2, Figure 3.1, Figure 2.11, Figure 2.8, Figure 2.5, and Figure 2.1.

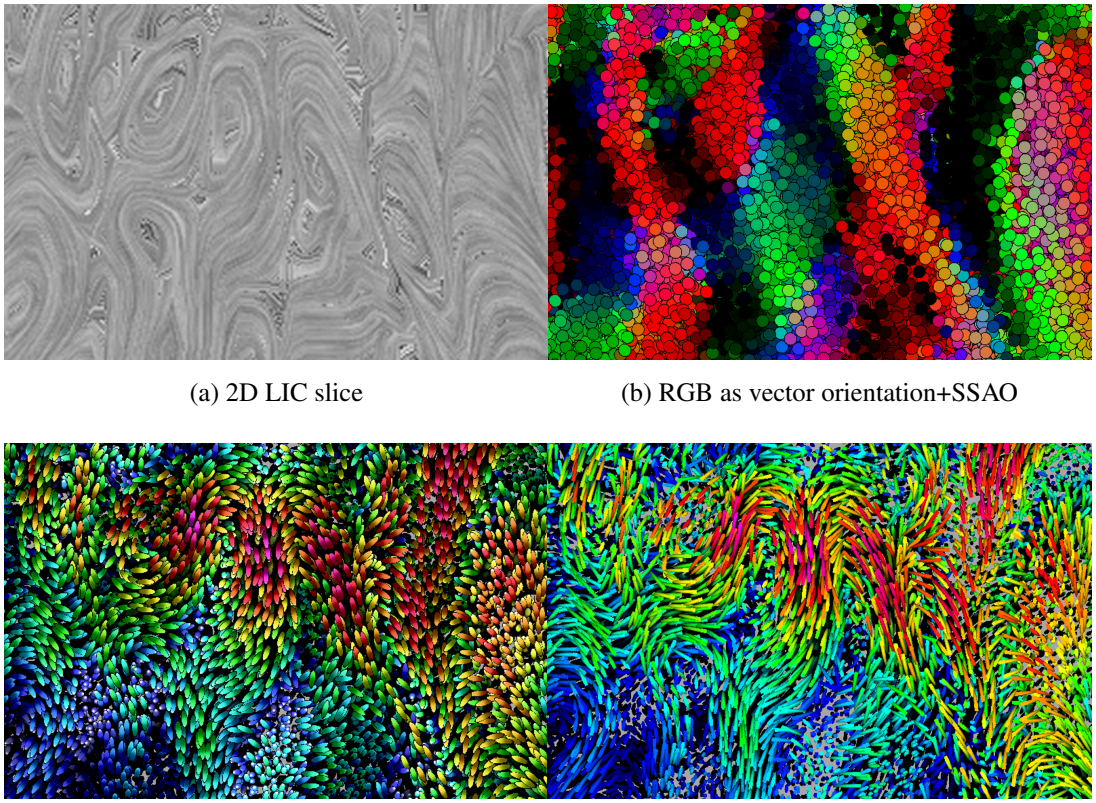
Although the simulation only saved the velocity field, GlyphSea computes displacements from the velocity data by time integration on the fly, and acceleration by temporal differentiation. A separate tool was created to compute curl and divergence of the fields. Because curl and divergence are only spatial operators and are not temporally dependent, temporal resolution has no effect on the output of this tool.

The scientists were able to observe the wavefronts clearly as well as the P- and S-waves initially when there were no boundary reflections. Later in the simulation, reflections can be visually tracked throughout the volume. With this simulation, the dipole texture applied to the ellipsoid and comet geometries were favored by scientists. The comet geometry was found to provide a LIC-like look that enhances visual comprehension of flow.

6.2 TeraShake Data Use Case

The second use case is from the publicly available TeraShake 2.1 dataset provided in the 2006 Visualization Contest. This data is an output for a simulation of 7.7 magnitude earthquake on southern San Andreas fault spanning a region of 600km x 300 km x 80km at a uniform grid of 750 x 375 x 100 voxels. The region of interest is a cropped region of that contains 100 x 125 x 20 voxels surrounding the Los Angeles basin for 75 timesteps. The Los Angeles basin is particularly interesting because it has many geophysical properties that are seen with vector visualization: wave guides, vorticity, reflection, and amplification.

In exploration it was easy to spot the wave guide effects and amplification in the Los Angeles basin. Surprisingly, vortices were found that were not exposed in prior scalar studies (see Table 6.1). Flow, such as the vortex features, can be seen with LIC and particle advection. However, LIC has an issue that it only looks at 2D vector orientation effectively. Because of this, 2D LIC is not capable of showing some of the vertical components of wave motion. For vector fields with disparate orientation continuity, LIC



(a) 2D LIC slice

(b) RGB as vector orientation+SSAO

(c) Ellipsoid+Dipole Texture+SSAO

(d) Comet+Dipole Texture+SSAO

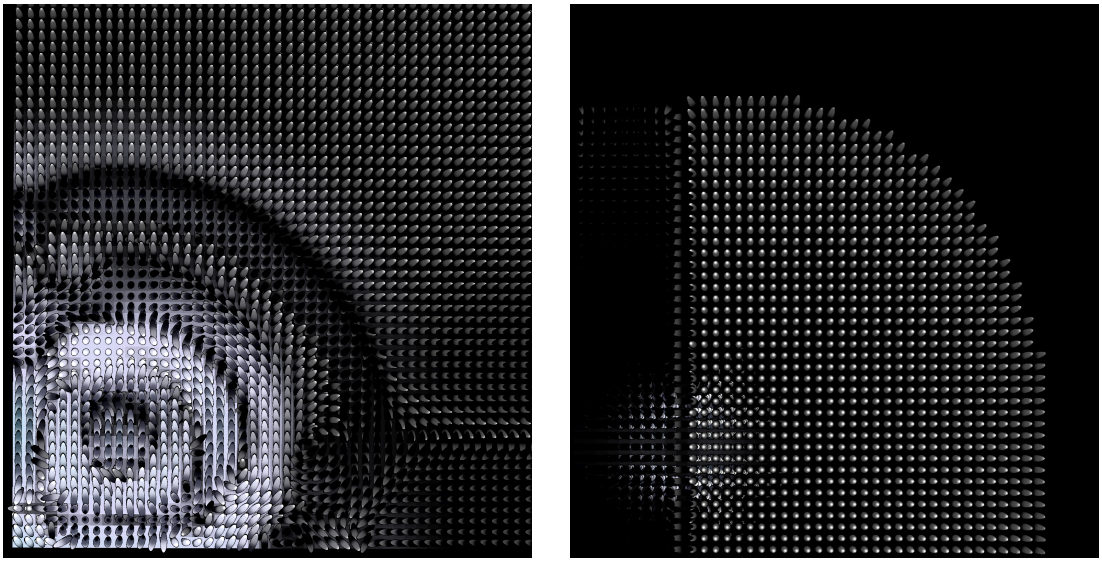
Figure 6.3: Comparison of previous vector methods (a,b) and novel glyph visualization techniques (c,d) applied to a TeraShake 2.1 volume crop. In (a) vector orientation is depicted by averaging noise along the vector direction, but only shows a 2D slice and is ambiguous with vectors orthogonal to the slice plane. In (b) the vector orientation is shown by mapping RGB colorspace to a normalized vector which is unintuitive, while GlyphSea's methods (c,d) use color to show magnitude and comet geometry with novel procedural dipole texturing to show orientation unambiguously. Screen space ambient occlusion (SSAO) has been applied to all glyph based methods (b,c,d).

would be difficult to understand, while glyphs retain their ability to display vector direction. This particular crop of TeraShake seems to have less orientation continuity than simple simulations. As seen in Figure 6.3, 2D LIC clearly indicates several vortices that can also be seen easily with the comet and ellipsoid dipole techniques. Closer analysis with the vector orientations show that the vortices continue through several depth slices, but diffuse to a more uniform wave motion beneath. The comet and ellipsoid glyphs indicate some upward motion, while this disappears with the LIC slice. Additionally, the comet and ellipsoid techniques show orientation, where 2D LIC must use glyphs to show orientation (as described in [Sha05]).

Contextual information is very important to correlate geologic features with wave propagation effects. The GlyphSea system includes several contextual cues described in Chapter 3 like a geographic map, sliprate of the fault, and an interactively modifiable isosurface of the ground characteristics. Isosurfaces allow for display of the sediment filled basins with an interactively tuned parameter to describe the surface created for ground stiffness. With TeraShake 2.1 the isovalue is configured to be 2.5 km/s, which is suspected to provide an isosurface of the ground stiffness volume that causes wave amplification in the Los Angeles basin. The isosurface is displayed in conjunction with volumetric velocity data shown by glyphs to provide context of the wave motion. With this contextual cue, wave guide effects are clearly seen in conjunction with amplification in the basin, and several eddies as a result of impacting the boundaries of this basin. Various parameters like glyph geometry, scale, and texturing are tuned interactively to highlight wave propagation.

GlyphSea is able to illustrate source directivity effects, wave guide and wave amplification features in context of 3D basin structure, and the conversion of P-waves to S-waves within TeraShake 2.1. GlyphSea is able to show vortices (see Table 6.1) in the velocity field data. These vortices were confirmed by computing and loading curl and divergence of the data within GlyphSea. For the TeraShake volume, scientists again favored ellipsoid and comet glyphs with dipole texturing which provides an intuitive sense of ground motion direction.

The performance of the system is interactive with moderate size data sets using procedurally generated glyphs. Although procedural generation may be slower than a



(a) Fortran (correct) (b) CUDA (incorrect)

Figure 6.4: This visualization clearly indicates that there is an error between the verified Fortran results (a) and CUDA results (b) even though L2-norm of error between fields is small. In this case 61^3 values are poorly compressed into a single floating point value that represents error, but visualizing magnitude and orientation is able to infer there are significant differences between the two.

texture lookup, it provides more precision than a lookup texture with limited orientations. This is useful for high resolution time dependent data to display fine changes in orientation during playback. In addition, there is no need to store high resolution texture lookups that would be necessary for our large format displays.

As the gap between memory bandwidth and compute bandwidth becomes larger on the GPU [AMHH08], procedurally generated glyphs could also become faster than texture lookups. At the very least, procedurally generated glyphs provide pixel accurate representations on high resolution and low resolution systems alike at interactive rates. Perhaps the largest problem with raycasting glyphs is aliasing at low resolutions.

6.3 Visualization for Error Purposes

Visualization is extremely helpful to verify simulations. Determining useful quantitative error metrics is difficult when one does not know the source of error. On the other hand, once a visualization system is in place, it can provide a very quick sanity check. Take an instance of porting a Fortran seismic simulation to CUDA. After the port was thought complete, traditional error metrics using absolute and relative error indicated a small amount of reported error only after many timesteps in the CUDA implementation. Due to the nature of floating point operations losing precision with roundoff, compounded with running floating point operations in different ordering [1] depending on thread scheduling on the highly parallel GPU with CUDA, the floating point computations are not expected to have the exact same results from the CPU fortran implementation and GPU implementation. Because of this, one cannot simply compare bits for equivalence.

Instead, this system used two comparisons: an absolute and relative hybrid [2] L2-norm of the difference between Fortran and CUDA, and a unit of least precision (ULP) error metric. L2-norm of the difference between the Fortran and CUDA simulation was approximately 0.002 for the velocity mesh, but a more robust error metric using ULP comparisons was reporting several significant differences. This motivated the decision to visualize the output, because it is extremely difficult for error metrics to do a full qualitative comparison and pinpoint the effect of the error.

Initially a hypothesis was formed to describe the error: a subtle thread issue may be causing incorrect synchronization and giving poor performance. Visualization indicated a drastic difference in orientation that was not indicated by either ULP or L2-norm Figure 6.4, showing this to be invalid. Further analysis indicated a typographical coding error where some of the vector fields were not being updated properly between timesteps.

Error metrics map a many dimensional space onto a smaller dimensional space without using features of the data (such as vector orientation) and consequently suffer in

[1] The order that floating point operations are executed can produce different results.

[2] A hybrid solution is needed because sometimes the value being compared is 0, which breaks a purely relative difference.

interpretation. In contrast, visualizing the full data does not have this problem, but does not provide a handy single value to test for accuracy.

6.4 Performance

In order to maintain a framerate f, then the total operations for a volume must complete in $1/f$ seconds. If the time is partitioned into time spent on the CPU t_c and GPU t_g, then framerate is $1/(t_c + t_g)$. Getting good performance calls for a careful balance between reducing t_c and t_g. If $t_g \ll t_c$, then it does not make sense to put effort into optimizing on the GPU. This was the case with early prototypes, and functions were rewritten to provide better cache locality by reducing loop complexity and iterating over memory sequentially as much as possible. Benchmarks on the current system show that achieving good performance with GlyphSea will require a careful balance between CPU optimizations and GPU optimizations.

It may be possible to reduce t_c to almost 0 in the situation where simulation is performed on the GPU at the same time as visualization. In this case, the CPU only needs to send initial conditions to the GPU. It is in essence like a compression technique where only minimal data is sent to reconstruct the simulation. An initial port of a subset of SORD [3], a seismic simulation created by Dr. Geoffrey Ely, indicated a promising order of magnitude performance increase over a single core Xeon. Maximum theoretical performance for medium sized datasets is within the realm of interesting real time simulations and visualization, but achieving this theoretical performance required more work than would fit within the focus of this work.

6.5 Benchmarks

The following subsections describe the benchmark methodology and present crafted graphs and analysis showcasing the bottlenecks currently within GlyphSea.

[3] `http://earth.usc.edu/~gely/sord/doc/doc.html`

6.5.1 Benchmark Methodology

Throughout performance analysis, there will be references to two machines. Comparing and contrasting performance between the two can help discover bottlenecks; however, it is also true that the different machines have different bottlenecks.

Machine "Quadro"	
CPU	7130M - 3.2 GHz Xeon 8k Data Cache (L1) - 2 MB L2 - 8 MB L3
Memory	4 GB DDR2 667 MHz
BUS	800 MHz FSB, PCIe 16x 1.0
GPU	Quadro FX 5800 - G200 based architecture 240 cores 4GB GDDR3 VRAM
Kernel	Linux Kernel 2.6.18-194.3.1.el5PAE
GPU Driver	NVIDIA UNIX x86 Kernel Module 256.35
GCC Version	4.1.2

Machine "GeForce"	
CPU	T9300 - 2.5 GHz Core 2 Duo 32k Data Cache (L1) - 6 MB L2 - no L3
Memory	2 GB DDR2 667 MHz
BUS	800 MHz FSB, PCIe 16x 1.0
GPU	GeForce 8600M GT - G80 based architecture 32 cores 512MB GDDR3 VRAM
Kernel	Mac OS X 10.5.8 Darwin Kernel 9.8.0
GPU Driver	NVIDIA driver 1.5.48
GCC Version	4.0.1

Benchmarks were calculated by determining the iterations of a draw loop that are calculated in 10 seconds. This is depicted in the following code block:

Typically frames per second is measured by looping over a set number of frames and calculating the time elapsed. Unfortunately, this can cause for extremely long benchmark runs when the framerate is small, and loss of precision when the framerate is extremely large. This is why the function iterates over time and exits once 10 seconds has elapsed. Each timing loop is performed twice in a row, with the first discarded. This is to alleviate slow sections that are common when a computer first encounters code, such as data not being stored in cache and branch prediction history not being built up.

```
double a = timer.seconds();
int frames = 0;
while ((timer.seconds() - a) < 10.0)
{
        ...
        frames++;
}
double b = timer.seconds();
double elapsed = (b - a);
double frames_per_second = frames / elapsed;
```

Compilation was performed with GCC given the flags "-O1 -ftree-vectorize". These flags gave the best performance while not causing a crash. Level O3 caused a crash, while O2 did not perform as well as O1 with autovectorization. Autovectorization is a technique implemented in GCC 4.0 that causes the compiler to automatically construct SIMD code using the SSE intrinsics for loops that appear vectorizable. Many of the loops are constructed simply and perform SIMD tasks. The core engine code achieves approximately a 9x speedup with optimization and autovectorization enabled as seen in Figure 6.5.

Testing was performed with vertical sync explicitly disabled. This is standard with testing performance with frames per second metrics because vertical sync will limit performance to the refresh rate of the monitor.

The square glyph performs the minimal steps required for a glyph splat, and every glyph using raycasting would be derived from this primitive and incur its overhead. It is useful for benchmarks by providing a maximum performance bound; the shader code for each glyph cannot perform better than for this square glyph. For these reasons the square glyph is used as a baseline for performance, and more complex glyphs with larger sizes are compared.

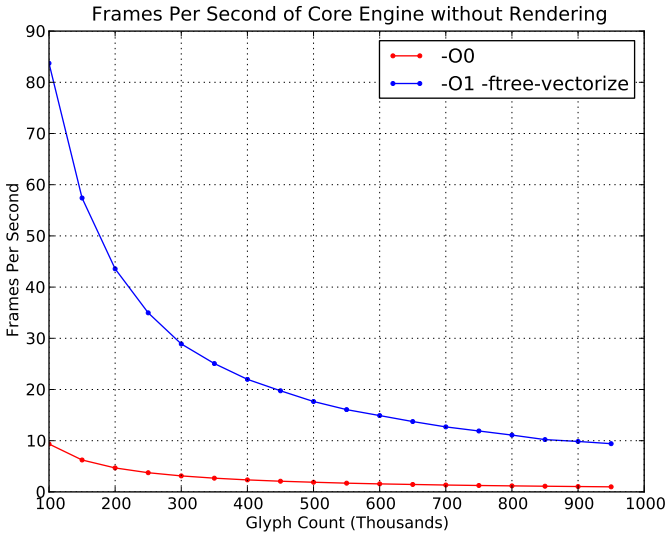


Figure 6.5: GCC optimization ("-O1 -ftree-vectorize") has drastic effects on performance, achieving between 8.95x and 9.54x speedup compared to the unoptimized binary ("-O0"). This was performed on machine "Quadro".

6.5.2 Benchmark Results

Figure Figure 6.6 depicts performance of the various glyphs with a static scene with machine GeForce, while figure Figure 6.7 depicts performance of the glyphs with machine Quadro. Because the scene is static, it causes for less operations to run than on a dynamic, playing scene. Glyph size does not effect runtime for the square glyph for radius 0.1, 0.5, 1.0, and 3.0, so only one line is drawn for the maximum of all four values.

For the lesser graphics card in system GeForce, pixel shader performance is a bottleneck to optimal performance. This is evidenced by the fact that increasing glyph radius, which would increase the number of fragments drawn per glyph, causes the performance to become worse relative to the baseline square glyph. On the other hand, machine Quadro is able to maintain performance close to the baseline, which indicates that for machine Quadro pixel shader performance is not a bottleneck. This is perhaps due to only having 32 cores in the GeForce GPU, while there are 240 cores in the Quadro GPU. Overall framerate decreases to less than 10 frames per second with large volumes

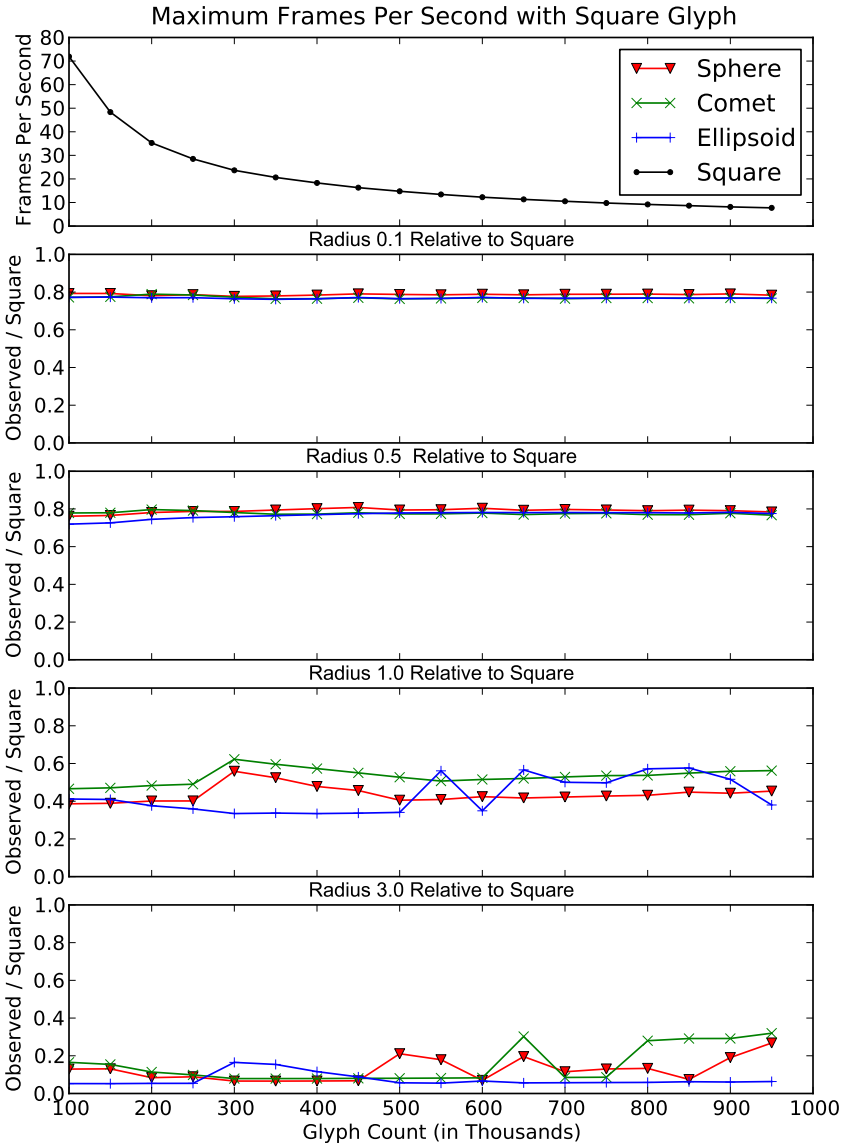


Figure 6.6: Performance with a GeForce 8600M GT of various glyphs with dipole shading. The topmost graph depicts frames per second for the square glyph, while subsequent graphs depict the ratio performance that the more complex glyphs attain in relation to the minimalist square glyph. Note that when the ratio is linear (seen in Radius 0.1), this implies that the runtime for drawing more complex glyphs are a constant multiple of the runtime for the basic square glyph.

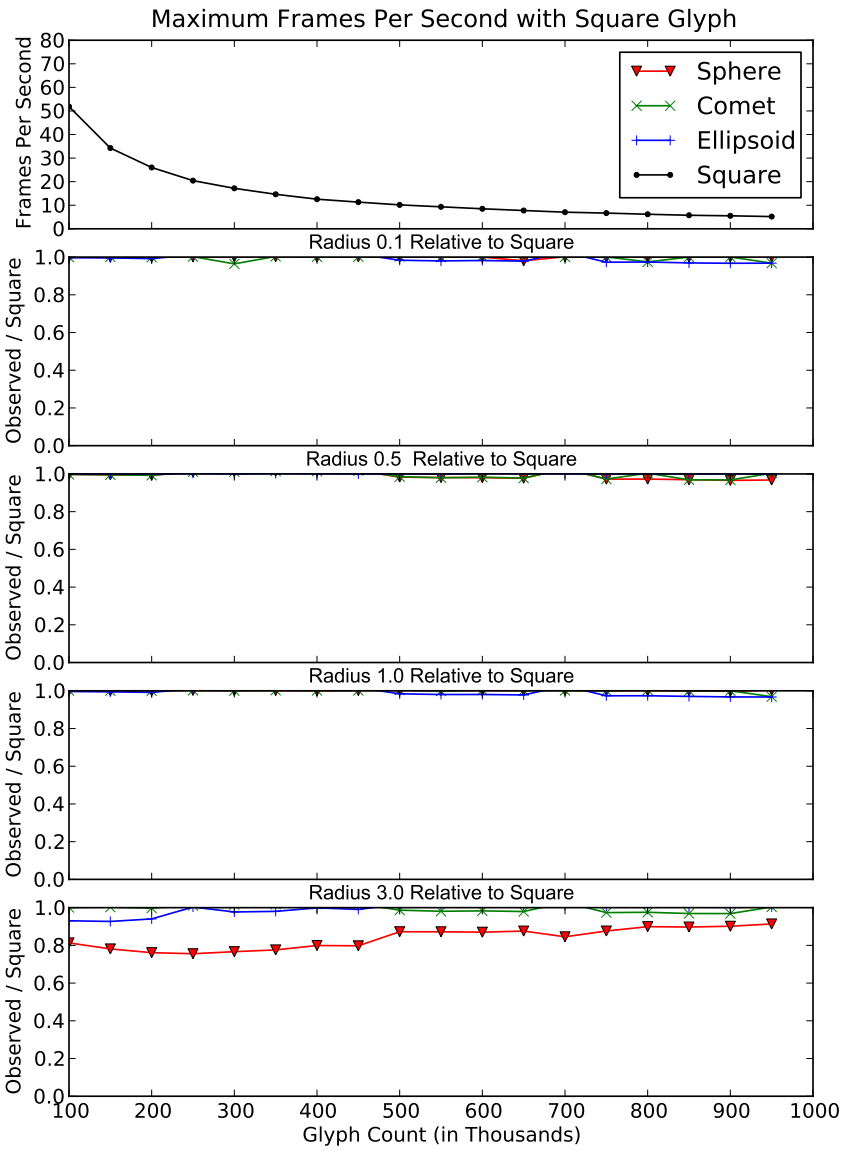


Figure 6.7: Performance with a Quadro FX 5800. Note that the Quadro is able to attain nearly the same performance across all glyphs, except when the radius becomes quite large.

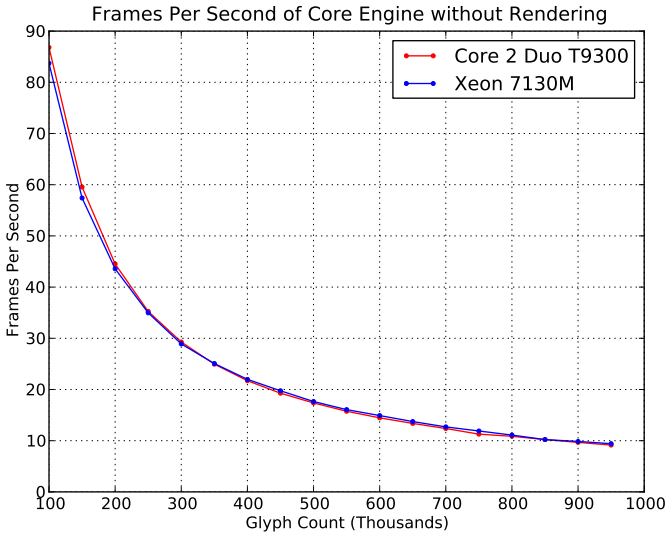


Figure 6.8: Both the machine "Quadro" (Xeon 7130M) and the machine "GeForce" (Core 2 Duo T9300) exhibit similar performance for the core engine without rendering. This system does not create graphical output for glyphs and does not copy any data to the GPU for glyphs, but otherwise includes the OpenScenGraph rendering traversal process. Without improvements to the core engine, rendering performance will not surpass these bounds.

using a square glyph, indicating that simply improving pixel shader performance would not achieve interactive rates with large volumes.

Notice that in Figure 6.6, the comet glyph is able to outperform the sphere glyph, but the sphere glyph outperforms the ellipsoid glyph. While the comet glyph geometry is a skinny ellipsoid, it performs better because there are fewer visible pixels visible given the same radius as the sphere or ellipsoid. The discarded pixels within the bounding square result in fewer calculations to calculate lighting across the glyph, explaining why the comet has advantageous performance.

Despite drastically different CPU architectures, both machines have nearly the same core engine performance (see Figure 6.9). This is fairly significant, as both machines have differing clockrates, cache designs, and SIMD intrinsics, which are all significant effects of performance. However, both machines are similar in their memory architecture, both have the same speed 800 MHz front side bus and memory type. Anal-

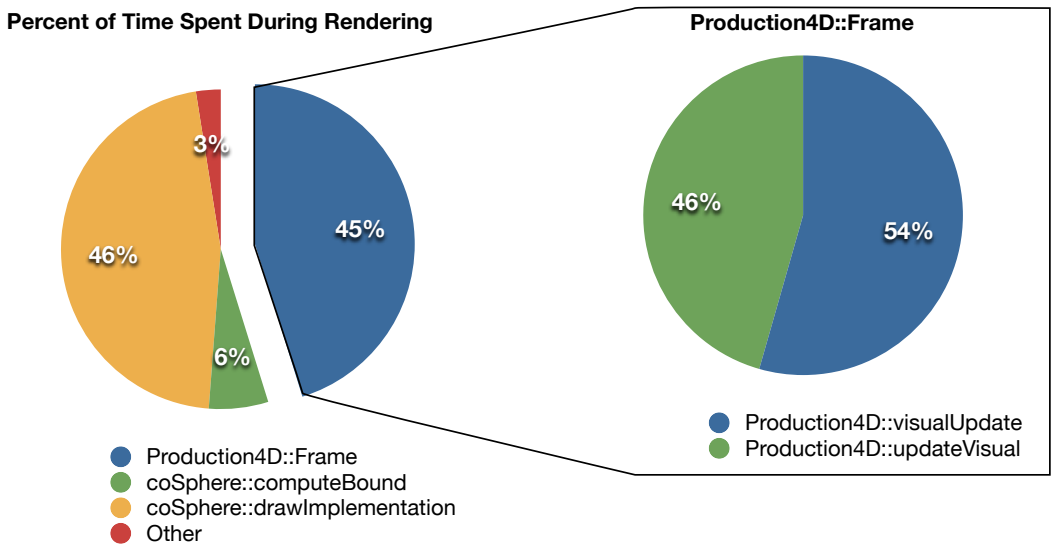


Figure 6.9: The Shark profiler tool (`http://developer.apple.com/tools/shark_optimize.html`) was used to determine a profile of GlyphSea rendering the TS 2.1 Usecase on machine "GeForce". "coSphere::drawImplementation" calls the OpenGL commands to render the glyph primitives, "coSphere::computeBound" is used by OpenSceneGraph to provide frustum culling and optimal near- and far-planes to prevent z-fighting, "Production4D::Frame" is part of the core engine which updates state for rendering. "Production4D::updateVisual" copies glyph state into a format used for rendering on the GPU, while "Production4D::visualUpdate" calls the filter calculations on the raw loaded data, which would include operations such as vector magnitude, calculating acceleration, or linear scaling that are sent to the GPU by updateVisual. Every one of these functions requires iterating over the glyphs, which creates a memory bottleneck.

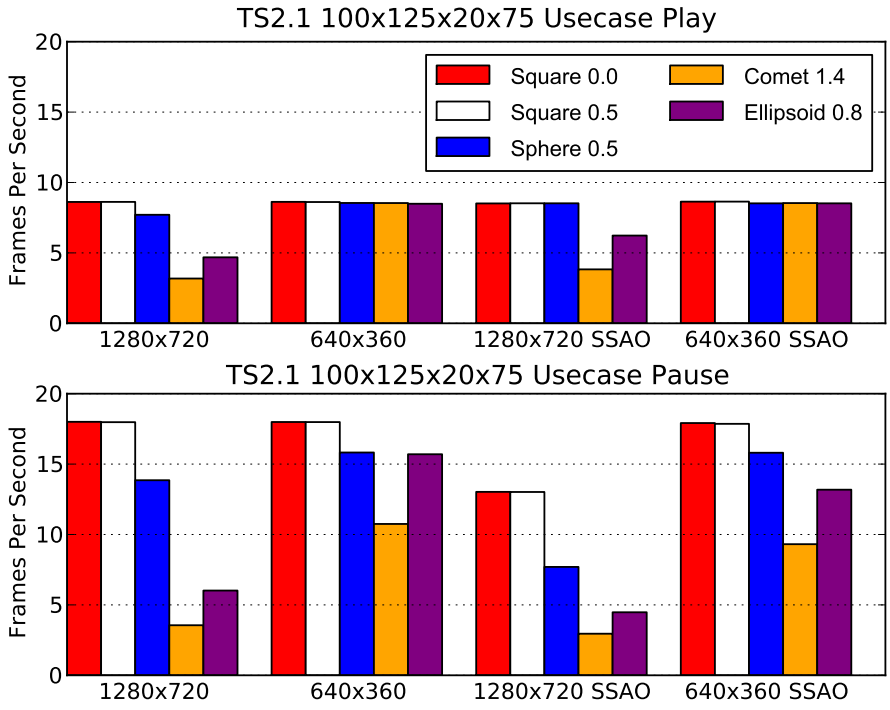


Figure 6.10: Performance is measured using a TeraShake 2.1 volume crop that is an interesting volume to seismologists with the "GeForce" system. This is performance with the fastest playback speed, but generally the volume playback is slowed down which yields better performance. With larger resolutions, the application becomes pixel fillrate bound. Unfortunately, the preferred ellipsoid and comet glyphs can drastically effect performance and indicate a bottleneck with the pixel shaders with this card.

ysis using a code profiler (see Figure 6.9) shows that the functionality of the methods using the most time is simply copying data. Combined with similar performance between CPUs with different computing architecture but similar memory architecture, this strongly indicates that performance is memory bound.

All of the operations that take considerable portions of time are simply streaming data through simple transformation kernels. As described in section 5.2.1, memory usage is significant with, in the worst case, 348 bytes per glyph. If every engine feature is enabled, it would require accessing and using each byte at least once. Reducing this cost and related memory copies should increase performance significantly.

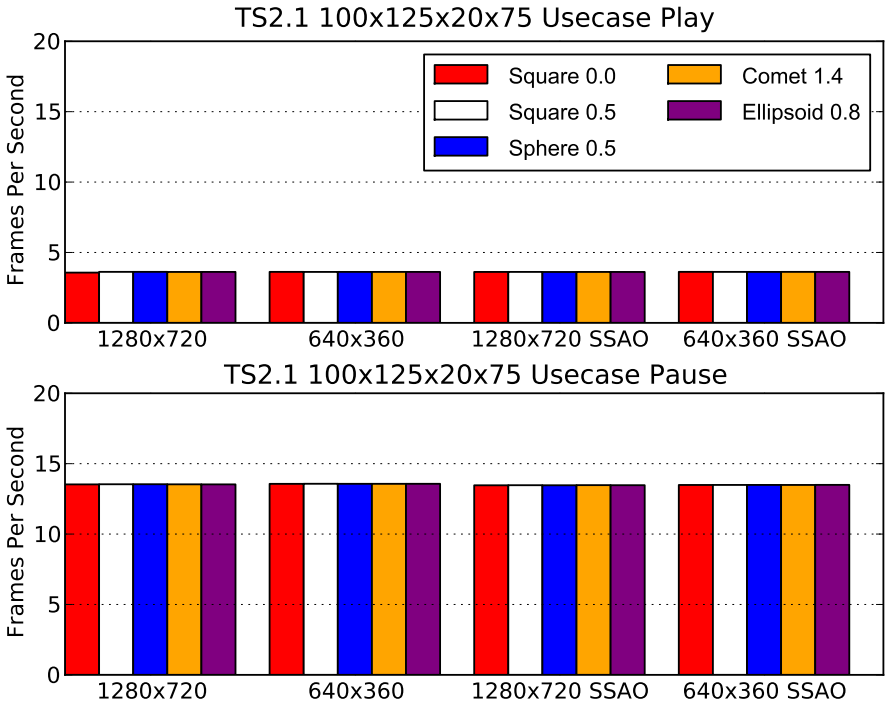


Figure 6.11: Performance is measured using a TeraShake 2.1 volume crop with the "Quadro" system. Because performance is so consistent, this provides evidence to conclude that the GPU is not a bottleneck. In comparison with the "GeForce" usecase, performance is worse during playback, but sometimes better when the volume is paused. Performance is consistent across different resolutions, indicating the system does not have a bottleneck with pixel fillrate.

Performance was measured for the TeraShake 2.1 100 x 125 x 20 x 75 crop described in section 6.2. Playback was set to the fastest rate, which causes the lowest framerate. Typical usecases does not actually playback at maximum framerate, which would give performance closer to the paused scene. Figure 6.10 depicts performance with machine GeForce, while Figure 6.11 depicts performance with machine Quadro. The results have different radii between the various glyphs, with each glyph given the size that has been determined visually useful after many iterations of rendering. The square with radius 0.0 is given as a case where there is no pixel shader overhead, and the resolution is changed to determine if the system is pixel fillrate limited.

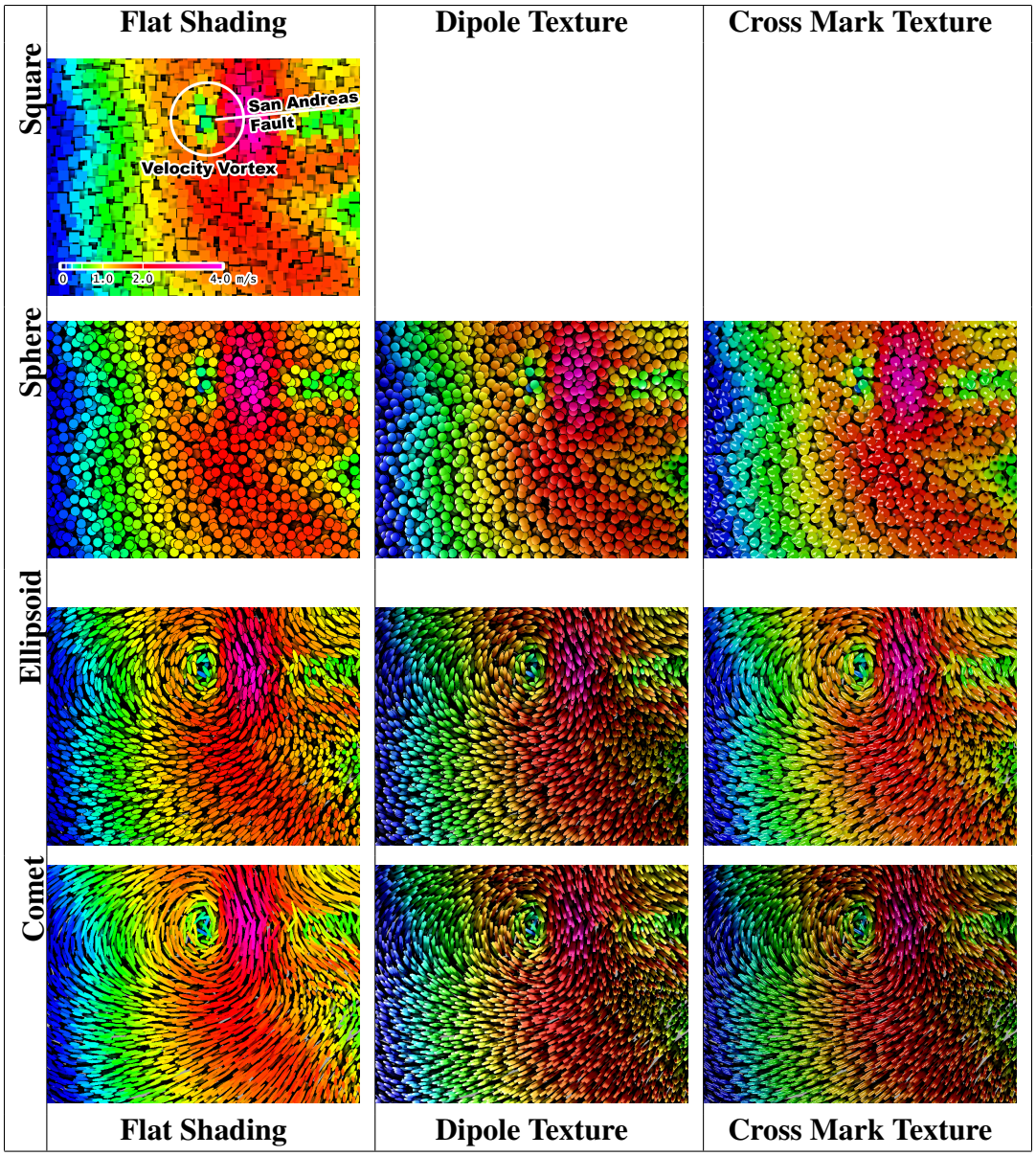
Interestingly, SSAO caused a repeatable slight increase in performance on machine GeForce (Figure 6.10). In the instance of the comet glyph, framerate with SSAO was ≈ 6.2 frames per second, while without SSAO framerate was ≈ 4.7 frames per second. It is not clear why this is the case, but is probably the result of graphics driver behavior. Subtleties in thread scheduling can result in significant differences in performance on the highly parallel GPU architectures, and it may be that including SSAO causes different scheduling that causes performance to increase slightly.

Machine Quadro has consistent performance under varying GPU changes that decrease performance on machine GeForce. This indicates the GPU is sufficient to saturate performance, and is able to render at high resolutions with SSAO using any of the common glyph settings without a decrease in performance. Although both systems have similar CPU engine performance (Figure 6.9), the Quadro system has worse rendering performance during playback. This would indicate that something between the core engine methods and the display methods is an additional bottleneck on Quadro. Unfortunately, this could be due to driver nuances between Linux and Mac OS X. Further analysis would be needed to determine this issue.

When the GPU is the bottleneck, the bottleneck appears to be related to pixel shader performance. In these cases either creating faster fragment shaders or reducing the number of primitives rendered will improve performance. When the CPU is the bottleneck, latency of more expensive GPU operations like SSAO or advanced glyph geometries is masked and does not cause a detriment in performance. CPU performance can be increased by reducing memory copies. The performance of both systems can be

increased by reducing memory copies, while increasing pixel shader performance would only currently effect systems with older graphics cards.

Table 6.1: Comparison of glyph geometry with different shading and texturing techniques. All glyphs are uniformly scaled with screen space ambient occlusion. Column one images are only able to show the magnitude of the velocity data, while column two and three images are able to show the velocity magnitude as well as the direction. The ellipsoid and comet glyphs in row three and four are able to show the wave flow better than others. The comet glyph in the bottom row is able to show the vortices highlighted by circle top left figure better than other glyphs.



Chapter 7

Conclusion

While scalar quantities are useful to provide surface data relevant to many, the elegance of visualization comes at a cost. The scalar quantities completely ignore relevant geophysics. It is haphazard at best to attempt to determine wave motion with scalar visualizations alone; in this work, hopefully vector visualization of seismic simulations is not only plausible but a useful alternative.

This thesis presents a set of techniques created to allow for interactive volumetric visualization of seismic fields using glyphs. Glyphs were chosen because of their ability to depict volumetric vector data intuitively compared to prior work with scalar volumetric data and colored volumetric fields. A novel technique of dipole texturing was developed to enhance glyph orientation. A combination of geometry, texturing, and contextual information is used to create a realtime interactive visualization environment that allows scientists flexibility in understanding data with improved comprehension.

Some techniques existed prior to this work, but other techniques are novel. Dipole texturing is perhaps one of the most significant contributions of this work that provides an unambiguous and view independent method to encode and show orientation information of vector data. Dipole texturing is widely applicable to any glyph geometry with non-trivial volume. A new technique of screen space ambient occlusion was implemented to provide an efficient depth enhancing method independent of scene geometry. This allows for ambient occlusion to be used regardless of scene complexity. A novel spacial context method was created with kelp-lattice and full-lattice to enhance perception of spatial neighborhoods with glyphs. Additional rich contextual multi-field

information is incorporated like maps, slices, and isosurfaces that allow rich environment for interactive exploration.

GlyphSea provides an extensive array of customizable parameters to enable rich and interactive exploration of seismic data. The proposed methods and techniques are not just confined to seismic domain; they are also applicable to other vector oriented data such as those produced by astrophysics and molecular dynamics simulations. GlyphSea is designed with novel and cutting edge techniques to push interactive vector visualization to provide a basis for future visualizations.

The focus of GlyphSea was not optimal implementation but rather one that is good enough to support small to medium volumes at interactive rates. Future work could undoubtably increase performance. Better performance can allow for larger volumes to be rendered interactively. However, the number of pixels in a screen is a limiting factor for intuitive visualization. With a minimum of approximately 10 pixels per glyph for good comprehension, a high resolution 2560x1600 pixel display would only be able to display a surface slice of 40960 glyphs. A volume of this size is in the realm of current interactive performance before significant optimizations, providing new and useful insight into the subtleties of wave propagation.

Appendix A

A Notational Note

Many fields have different notation for common linear algebra operation. Within computer graphics sometimes the magnitude of a vector is represented as $|\mathbf{v}|$ although a more common standard is $||\mathbf{v}||$. Using the latter notation is especially important when the notation for a vector is similar to a scalar to resolve confusion with the absolute value operator.

In statistics, a hat vector represents a vector of fit. Within linear algebra, orthogonal projections in some texts are denoted by a hat vector as well as a unit vector. In many computer vision circles, a hat vector corresponds with the skew symmetric representation of the cross product for a vector.

There are many more situations where notation is not precise, even within similar contexts, and even changes over time. To combat the sands of time, notation used throughout the thesis is described here:

- Scalars are noted as a single-letter lowercase variable s
- Vectors are noted as a single-letter lowercase variable $\mathbf{v}$ and are either 3-component coordinates or 4-component homogeneous coordinates.
- Vector components are noted as $\mathbf{v} = \langle v_x, v_y, v_z \rangle$.
- Absolute value is represented as $|s|$ while vector magnitude $||\mathbf{v}||$.
- Gradient of a field is ∇, cross product is $\mathbf{v_1} \times \mathbf{v_2}$, and dot product is $\mathbf{v_1} \cdot \mathbf{v_2}$.

- Matrices are noted as a single-letter capital variable M and are typically $\in \mathbb{R}^{3\times3}$ or $\in \mathbb{R}^{4\times4}$.
- Because every variable is single-letter, there is implicit multiplication between letters. For instance, $\mathrm{RR}^{\mathrm{T}}\mathbf{v}s$ will multiply the scalar s times the vector $\mathbf{v}$ times the matrix transpose R^{T} times the matrix R.
- Vector addition and subtraction is componentwise.

Appendix B

Benchmark Tabular Results

(f) is frames per second (r) is relative to square glyph

B.1 Quadro Engine Performance

Glyph Count	Square Radius 3.0 (f)
12500	606.00364542
50000	167.57184793
100000	83.7211346911
150000	57.4012125133
200000	43.5678904647
250000	34.9725814961
300000	28.900236384
350000	25.0668115415
400000	21.9853138104
450000	19.7499142175
500000	17.6550326319
550000	16.0683453596
600000	14.9066989711
650000	13.7390672368
700000	12.6953281193
750000	11.8972755239
800000	11.0964269505
850000	10.2162573386
900000	9.85122657722
950000	9.41733463656

B.2 GeForce Engine Performance

Glyph Count	Square Radius 3.0 (f)
12500	778.544723325
50000	177.763380744
100000	86.8123195572
150000	59.5512275446

200000	44.5146653865
250000	35.2470589175
300000	29.2792410181
350000	24.9520172708
400000	21.7182741344
450000	19.2769255202
500000	17.3755352464
550000	15.7504019842
600000	14.4633642982
650000	13.374134424
700000	12.3811187938
750000	11.2804509785
800000	10.8638992627
850000	10.2194807115
900000	9.66993616845
950000	9.14274811133

B.3 GeForce Engine Performance -O0

Glyph Count	Square Radius 3.0 (f)
12500	74.4153153711
50000	18.8115481008
100000	9.3550489896
150000	6.23295830052
200000	4.68001633027
250000	3.74516331869
300000	3.12467654715
350000	2.68035833413
400000	2.3433243571
450000	2.08551609075
500000	1.88605451293
550000	1.7062371499
600000	1.56360857405
650000	1.45100394963
700000	1.3406841128
750000	1.25091051852
800000	1.17592528171
850000	1.10800050229
900000	1.04731881623
950000	0.987485595054

B.4 GeForce Engine Performance -O1

Glyph Count	Square Radius 3.0 (f)
12500	606.00364542
50000	167.57184793
100000	83.7211346911
150000	57.4012125133
200000	43.5678904647
250000	34.9725814961
300000	28.900236384
350000	25.0668115415
400000	21.9853138104
450000	19.7499142175

500000	17.6550326319
550000	16.0683453596
600000	14.9066989711
650000	13.7390672368
700000	12.6953281193
750000	11.8972755239
800000	11.0964269505
850000	10.2162573386
900000	9.85122657722
950000	9.41733463656

B.5 Quadro Glyph Performance for Radius 0.1

Glyph Count	Sphere Radius 0.1 (r)	Comet Radius 0.1 (r)	Ellipsoid Radius 0.1 (r)	Square Radius 0.1 (f)
12500	0.998598733884	0.994696118165	0.994723966343	368.203162568
50000	0.998944329823	0.99886049102	0.998901410582	102.569947006
100000	1.00099951131	0.999972020566	0.996565693195	51.6997415013
150000	1.0005384454	1.00097942312	0.995499314809	34.757561018
200000	0.99982437969	0.99905975398	0.991847173522	25.8327573327
250000	1.0013955325	1.00225535699	1.0122535241	20.9713531316
300000	1.00092877743	0.96402381534	1.00990166366	17.3642470154
350000	1.00194294649	1.00348693397	1.01483301069	14.8168183816
400000	0.999791049146	1.00103010928	1.01488924666	12.7315804865
450000	1.0013554009	1.00184326157	1.01568560561	11.4761983646
500000	0.999915599919	1.00350171249	0.983220682448	9.93988358408
550000	1.00004457877	0.999631634296	0.979672513133	9.12833345405
600000	1.00005585929	1.00457240533	0.981855911619	8.30707703627
650000	0.980337664438	1.00083616446	0.978499954313	7.56917830594
700000	1.00180158337	1.00086868035	1.02561553	7.21753247395
750000	0.99877901091	1.00122098663	0.972828729519	6.48746621527
800000	1.00052513668	0.974399890645	0.973881147457	5.99768489363
850000	1.00008204946	1.00020465445	0.969113333629	5.55848368535
900000	1.00059517709	1.00049363156	0.967165987378	5.33065879045
950000	1.00109949533	0.967886421188	0.967838739236	5.01932439894

B.6 Quadro Glyph Performance for Radius 0.5

Glyph Count	Sphere Radius 0.5 (r)	Comet Radius 0.5 (r)	Ellipsoid Radius 0.5 (r)	Square Radius 0.5 (f)
12500	0.998464945869	0.995123953685	0.994562715598	368.239976884
50000	0.996879333996	0.99640646598	0.99822095714	102.723060428
100000	0.997949559408	0.997286914268	0.999555512051	51.8612078166
150000	0.995343100305	0.9954505547	0.999320528809	34.9397987268
200000	0.992816735622	0.992705719328	0.99898305693	26.0254111716
250000	1.01075859521	1.01163377273	1.00310436026	20.7873405096
300000	1.01217266943	1.01208789007	0.999659249191	17.1654802193
350000	1.01454425985	1.01499260088	1.00288376331	14.6382558369
400000	1.01448376987	1.01458863827	1.00171270261	12.5563541131
450000	1.01754762165	1.01729252331	1.00033384443	11.3133285897
500000	0.983154102677	0.985284221005	1.00143100925	10.1415239728
550000	0.980931119708	0.979771605962	0.999339709214	9.31202925958
600000	0.980124090884	0.982871813389	1.00251384666	8.47702725614
650000	0.977335767544	0.978050136569	1.00182126884	7.75531388142
700000	1.0273873783	1.02662624885	1.00109826383	7.0377022596
750000	0.972532865295	0.972966537566	0.999333003978	6.66286121161

800000	0.972719590851	1.00392865992	1.00185842203	6.18650722774
850000	0.969485578601	0.968912420956	1.0002115766	5.73224486569
900000	0.96593043328	0.968051420085	1.00049560312	5.4988452425
950000	0.96784732164	1.0039750565	1.0003885454	5.19353923719

B.7 Quadro Glyph Performance for Radius 1.0

Glyph Count	Sphere Radius 1.0 (r)	Comet Radius 1.0 (r)	Ellipsoid Radius 1.0 (r)	Square Radius 1.0 (f)
12500	0.999179009025	0.997404404762	0.994103167381	367.98160092
50000	0.998092271698	1.00193161979	0.996833521023	102.427993121
100000	1.00138541551	1.00106247536	0.996275859812	51.6732849117
150000	0.999575845762	1.00061141487	0.994473375312	34.7686734252
200000	1.00007984781	1.00007086429	0.991387319316	25.8484065805
250000	1.00229927012	1.0025438568	1.01183120458	20.9565779704
300000	1.00093976149	1.00088184829	1.01111294001	17.3434776065
350000	1.00143831042	1.00265189075	1.01553740353	14.8265050146
400000	0.999999004792	1.00158888855	1.01586796086	12.7381816047
450000	1.00131170542	1.00176075887	1.01636975053	11.5049575457
500000	1.00162524158	1.00398293966	0.984035055476	9.97141196191
550000	1.00157653747	0.998567969917	0.98027320647	9.14662941735
600000	1.00099243097	1.0040265188	0.980367516606	8.31238743642
650000	1.00163884348	1.00150430997	0.978130117293	7.56898231049
700000	1.00191010387	1.001296128	1.02612724327	7.22389625297
750000	1.00023281312	1.00027162585	0.973552171705	6.47924697195
800000	1.00065001263	1.00367436449	0.973809009894	6.0171973473
850000	0.999811238522	1.00004942577	0.97022610851	5.55097930179
900000	0.997096934852	1.00034184464	0.967599895534	5.32901222811
950000	1.00365819846	0.968690091593	0.967154544147	5.02069807392

B.8 Quadro Glyph Performance for Radius 3.0

Glyph Count	Sphere Radius 3.0 (r)	Comet Radius 3.0 (r)	Ellipsoid Radius 3.0 (r)	Square Radius 3.0 (f)
12500	0.913127834953	0.995773663383	0.988392379364	367.675733402
50000	0.843138532534	1.00120593631	0.92025028732	102.568203857
100000	0.813192320037	1.00077170759	0.930525183887	51.857425054
150000	0.781202807354	1.00261589095	0.926848822281	34.259265733
200000	0.76133061059	0.997695017336	0.940236249138	26.0195475589
250000	0.755676727598	1.01108194807	1.00321833598	20.4367482641
300000	0.766595286897	1.0159917803	0.977138616099	17.1739642702
350000	0.775989945325	1.01792463265	0.980297271628	14.6455040793
400000	0.799299522667	1.01793560964	0.998853835727	12.5641294105
450000	0.79769307284	1.02022730018	0.990691129246	11.3208446542
500000	0.872326395219	0.98671699573	1.01771092182	10.152354998
550000	0.87202772032	0.980827897423	1.01402140276	9.32748810745
600000	0.869966071692	0.982998583215	1.02148557458	8.49330727387
650000	0.8758774458	0.979370515436	1.01199915022	7.76135620744
700000	0.845418434885	1.02856078847	1.01256387845	7.04678969104
750000	0.876141931873	0.973730040936	1.00845533447	6.66691874917
800000	0.899208765786	0.97552203246	1.00585737735	6.17630153103
850000	0.89732516715	0.968879744476	1.00307267029	5.73827339311
900000	0.901444330733	0.968586715934	1.0014300113	5.51067742954
950000	0.913621068755	1.00358024849	1.0042209814	5.18697550451

B.9 GeForce Glyph Performance for Radius 0.1

Glyph Count	Sphere Radius 0.1 (r)	Comet Radius 0.1 (r)	Ellipsoid Radius 0.1 (r)	Square Radius 0.1 (f)
12500	0.725967083533	0.848291094932	0.847867637701	599.474822057
50000	0.787419857819	0.76811750318	0.767360192343	137.184635321
100000	0.792732675272	0.771679121025	0.771978161713	69.472072227
150000	0.792560437772	0.774783634665	0.773253347448	46.6666799905
200000	0.780426408385	0.788964030612	0.769731970501	34.3622015783
250000	0.786221718625	0.784518364885	0.770221402854	27.690945061
300000	0.777101368832	0.77259474344	0.764484434132	23.1941086964
350000	0.779351257513	0.764268468084	0.761398466983	20.1568643104
400000	0.783984912949	0.764905814139	0.764315905019	17.7051358819
450000	0.790382957883	0.769565112307	0.771066531677	15.6939421383
500000	0.787391803736	0.763801388409	0.765168246263	14.1935277513
550000	0.785389509939	0.766288550123	0.765962642431	12.8783257777
600000	0.7883849074	0.769075824998	0.771634446858	11.7622197502
650000	0.785307151688	0.768661315154	0.766600526081	10.8898941782
700000	0.788148728815	0.764989877913	0.76722208371	10.1238283158
750000	0.788413141642	0.7684523535	0.766809076917	9.41512266914
800000	0.789229438682	0.768193259353	0.767734856559	8.84253238205
850000	0.786726085236	0.766723199357	0.76758834192	8.32152799104
900000	0.790149680904	0.768928834139	0.767324720666	7.81782682227
950000	0.783433860401	0.766344894832	0.767639384842	7.43097368544

B.10 GeForce Glyph Performance for Radius 0.5

Glyph Count	Sphere Radius 0.5 (r)	Comet Radius 0.5 (r)	Ellipsoid Radius 0.5 (r)	Square Radius 0.5 (f)
12500	0.644934934771	0.739365597663	0.54997228376	599.312500375
50000	0.782900908834	0.755820288514	0.658660157201	137.373761612
100000	0.76133460304	0.778488302754	0.719269622718	69.690870496
150000	0.765323752195	0.779620841261	0.726006532332	46.6827273909
200000	0.780916266914	0.796574510571	0.744725571684	34.3910583248
250000	0.786718463589	0.791265524332	0.754069160951	27.7367601868
300000	0.786779956842	0.780815510225	0.758606781998	23.1656453479
350000	0.794098677166	0.771785698076	0.764121001189	20.1580712119
400000	0.801416403209	0.772565956032	0.769486181482	17.675731221
450000	0.807881709318	0.779626211847	0.774691787946	15.6717594894
500000	0.794137913919	0.773422458703	0.778441189148	14.2189378354
550000	0.795698368257	0.773719722795	0.779923709508	12.9138003824
600000	0.803503264174	0.778058208753	0.781125985651	11.786316087
650000	0.792778668347	0.770016443371	0.780313072163	10.9160230348
700000	0.797013195639	0.774913632327	0.780851394358	10.1478905819
750000	0.794401546484	0.776555760816	0.780197611819	9.4324073688
800000	0.790557165059	0.769602377535	0.779938623509	8.86949779708
850000	0.793514076144	0.769379358899	0.778323655539	8.35624669232
900000	0.790693718895	0.777030877968	0.78195720622	7.85268755747
950000	0.783846120051	0.766808925812	0.775479767557	7.4727990116

B.11 GeForce Glyph Performance for Radius 1.0

Glyph Count	Sphere Radius 1.0 (r)	Comet Radius 1.0 (r)	Ellipsoid Radius 1.0 (r)	Square Radius 1.0 (f)
12500	0.322018749437	0.373479314211	0.373764109904	599.546040856
50000	0.378351112296	0.452016091598	0.458396685401	137.223429326

100000	0.386161867118	0.466029039579	0.411643847924	69.6202847739
150000	0.388939040595	0.470860631447	0.409152586744	46.5983690571
200000	0.401102669967	0.483032920514	0.375951227774	34.3641925114
250000	0.401389588498	0.490204804291	0.359238162689	27.7158270333
300000	0.55874786063	0.622514946774	0.334545862158	23.1779577622
350000	0.524405409734	0.596023330498	0.337074096495	20.1495254388
400000	0.478142462485	0.573061177195	0.334387175679	17.7273003413
450000	0.45670163693	0.550286976736	0.336721490002	15.6900211466
500000	0.405315136935	0.527134830497	0.34065900887	14.1889042769
550000	0.408972262851	0.506876233379	0.561057021571	12.8948033942
600000	0.423989783189	0.515229872716	0.349108302794	11.7471144906
650000	0.417084891422	0.520453836901	0.566035578761	10.8762679833
700000	0.422115424286	0.528680221309	0.500305179064	10.1234464976
750000	0.4272805493	0.53574673615	0.497512105228	9.39757542554
800000	0.431404570332	0.536829050353	0.57168563306	8.82381517974
850000	0.448276286253	0.54897823256	0.576187418887	8.33414359729
900000	0.442242646119	0.558965294158	0.516202838962	7.81761794213
950000	0.453582801777	0.562228306563	0.380039263719	7.44510722443

B.12 GeForce Glyph Performance for Radius 3.0

Glyph Count	Sphere Radius 3.0 (r)	Comet Radius 3.0 (r)	Ellipsoid Radius 3.0 (r)	Square Radius 3.0 (f)
12500	0.152617877045	0.178202354387	0.0489268950201	496.886584062
50000	0.175818961484	0.210451852193	0.0527359998615	137.711176291
100000	0.129464987653	0.165339007956	0.052461575342	71.913559901
150000	0.130736367256	0.15417679791	0.0522739971343	48.3456111874
200000	0.0839752164348	0.113888392578	0.0535702754519	35.2752720343
250000	0.0885447018235	0.0986047118193	0.0537932058489	28.4975207157
300000	0.0659785847793	0.0790473888661	0.164958371768	23.6496734948
350000	0.0657413249919	0.0791212571398	0.154077471217	20.6341358384
400000	0.0662476607264	0.0792663258327	0.116043867301	18.3038134603
450000	0.0673413515334	0.0802947724883	0.0878872784289	16.2970991164
500000	0.210917795076	0.0807106316593	0.0567521627219	14.7599562387
550000	0.179038119083	0.0817596180221	0.0548859479003	13.4156424403
600000	0.0695788781008	0.0830251652171	0.0661512462193	12.2501907245
650000	0.195625982589	0.302172564554	0.0562571187169	11.3279992369
700000	0.11523334354	0.0850515755487	0.0571537334971	10.5214677646
750000	0.130076280467	0.0863699712835	0.0580679476866	9.77830194798
800000	0.133074385392	0.279831591923	0.058720933726	9.18033572089
850000	0.0744151532055	0.291543934841	0.0624722593063	8.65844810753
900000	0.190420950622	0.291696247007	0.060826563098	8.12947678886
950000	0.267864976964	0.319761647162	0.0634241268454	7.72656670999

B.13 GeForce Usecase No Play

	Square 0.0 (f)	Square 0.5 (f)	Sphere 0.5 (f)	Comet 1.4 (f)	Ellipsoid 0.8 (f)
HD	18.0026954312	17.977635821	13.8538252006	3.5546041504	6.01830155637
QHD	17.9892064761	17.9845872088	15.8227375724	10.74815092	15.6988696814
HD SSAO	13.0274112644	13.0193063393	7.69641347132	2.95324423724	4.47574593279
QHD SSAO	17.9154926213	17.8588175667	15.8054674989	9.31179864325	13.181664305

B.14 GeForce Usecase Play

	Square 0.0 (f)	Square 0.5 (f)	Sphere 0.5 (f)	Comet 1.4 (f)	Ellipsoid 0.8 (f)
HD	13.5277213846	13.5376186532	13.5397616006	13.5327557367	13.528959434
QHD	13.5666666999	13.5724885657	13.5688865432	13.5686157917	13.5699425772
HD SSAO	13.4635004503	13.469908225	13.4655013855	13.4754073815	13.4706609006
QHD SSAO	13.4871602235	13.48932994	13.4898286692	13.4885347455	13.4938468059

B.15 Quadro Usecase Play

	Square 0.0 (f)	Square 0.5 (f)	Sphere 0.5 (f)	Comet 1.4 (f)	Ellipsoid 0.8 (f)
HD	3.56956109834	3.62255343022	3.62340839338	3.61673551497	3.6192013303
QHD	3.6218903135	3.61815374415	3.62029911498	3.61996616799	3.61980680015
HD SSAO	3.61776105568	3.61909158845	3.61509232653	3.61730833154	3.61699361356
QHD SSAO	3.62337645804	3.62100771666	3.62056835095	3.62214560242	3.62102543526

B.16 Quadro Usecase No Play

	Square 0.0 (f)	Square 0.5 (f)	Sphere 0.5 (f)	Comet 1.4 (f)	Ellipsoid 0.8 (f)
HD	8.6184265923	8.62180881585	7.70667219966	3.17989985303	4.68534891395
QHD	8.62480036561	8.6129745056	8.54615070441	8.54064253439	8.48997334148
HD SSAO	8.51292975447	8.52252175473	8.52155060513	3.82972079371	6.23085618887
QHD SSAO	8.63929169724	8.64191767133	8.5188747557	8.53904827342	8.51960896976

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[BKLW06] Ø. Bergmann, G. Kindlmann, A. Lundervold, and C.-F. Westin. Diffusion k-tensor estimation from q-ball imaging using discretized principal axes. In *Nineth International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI'06)*, Lecture Notes in Computer Science 4191, pages 268–275, Copenhagen, Denmark, October 2006.

[Bou94] Paul Bourke. Polygonising a scalar field, `http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/` 1994.

[BS09] Louis Bavoil and Miguel Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH '09: SIGGRAPH 2009: Talks*, pages 1–1, New York, NY, USA, 2009. ACM.

[CCC+07] Amit Chourasia, Steve Cutchin, Yifeng Cui, Reagan W. Moore, Kim Olsen, Steven M. Day, J. Bernard Minster, Philip Maechling, and Thomas H. Jordan. Visual insights into high-resolution earthquake simulations. *IEEE Computer Graphics and Applications*, 27(5):28–34, 2007.

[CCM09] Cheng-Kai Chen, Carlos Correa, and Kwan-Liu Ma. Frequency enhancements for visualizing 3d seismic data. Technical report, University of California at Davis, 2009.

[CCO+08] A. Chourasia, S. M. Cutchin, K. B. Olsen, B. Minster, S. Day, Y. Cui, P. Maechling, R. Moore, and T. Jordan. Visualizing the ground motions of the 1906 san francisco earthquake. *Computers and Geosciences*, 34:1798–1805, 2008.

[DDS+09] Thomas A. DeFanti, Gregory Dawe, Daniel J. Sandin, Jurgen P. Schulze, Peter Otto, Javier Girado, Falko Kuester, Larry Smarr, and Ramesh Rao. The starcave, a third-generation cave and virtual reality optiportal. *Future Gener. Comput. Syst.*, 25(2):169–178, 2009.

[DLR+09] Thomas A. DeFanti, Jason Leigh, Luc Renambot, Byungil Jeong, Alan Verlo, Lance Long, Maxine Brown, Daniel J. Sandin, Venkatram Vishwanath, Qian Liu, Mason J. Katz, Philip Papadopoulos, Joseph P. Keefe, Gregory R. Hidley, Gregory L. Dawe, Ian Kaufman, Bryan Glogowski, Kai-Uwe Doerr, Rajvikram Singh, Javier Girado, Jurgen P. Schulze, Falko Kuester, and Larry Smarr. The optiportal, a scalable visualization, storage, and computing interface device for the optiputer. *Future Gener. Comput. Syst.*, 25(2):114–123, 2009.

[EBRI09] Maarten H. Everts, Henk Bekker, Jos B. T. M. Roerdink, and Tobias Isenberg. Depth-dependent halos: Illustrative rendering of dense line data. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1299–1306, 2009.

[GGS01] Stefan Guthe, Stefan Gumhold, and Wolfgang Straßer. Texture particles: Interactive visualization of volumetric vector fields, 2001.

[GGS02] Stefan Guthe, Stefan Gumhold, and Wolfgang Straßer. Interactive visualization of volumetric vector fields using texture based particles. In *Journal of WSCG*, pages 33–41, 2002.

[GP06] Christiaan P. Gribble and Steven G. Parker. Enhancing interactive particle visualization with advanced shading models. In *APGV '06: Proceedings of the 3rd symposium on Applied perception in graphics and visualization*, pages 111–118, New York, NY, USA, 2006. ACM.

[Gum03] Stefan Gumhold. Splatting illuminated ellipsoids with depth correction. In *VMV*, pages 245–252, 2003.

[Hal86] Edmond Halley. An historical account of the trade winds, and monsoons, observable in the seas between and near the tropicks; with an attempt to assign the phisical cause of said winds. *Philosophical Transactions*, 183:153–168, 1686.

[KBKW07] Polina Kondratieva Kai Buerger, Jens Schneider, J. Krueger, and Ruediger Westermann. Interactive visual exploration of unsteady 3d flows. *Eurographics*, pages 251–258, 2007.

[Kin04] Gordon Kindlmann. Superquadric tensor glyphs. In *Proceedings of IEEE TVCG/EG Symposium on Visualization 2004*, pages 147–154, May 2004.

[KKW05] Polina Kondratieva, Jens Kruger, and Rudiger Westermann. The application of gpu particle tracing to diffusion tensor field visualization. *Visualization Conference, IEEE*, 0:10, 2005.

[KLRS04] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, New York, NY, USA, 2004. ACM.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[KW99] Gordon Kindlmann and David Weinstein. Hue-balls and lit-tensors for direct volume rendering of diffusion tensor fields. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 183–189, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[KW06] Gordon Kindlmann and Carl-Fredrik Westin. Diffusion tensor visualization with glyph packing. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006)*, 12(5):1329–1335, September-October 2006.

[LCD06] Thomas Luft, Carsten Colditz, and Oliver Deussen. Image enhancement by unsharp masking the depth buffer. *ACM Transactions on Graphics*, 25(3):1206–1213, jul 2006.

[LKJ+05] David H. Laidlaw, Robert M. Kirby, Cullen D. Jackson, J. Scott Davidson, Timothy S. Miller, Marco da Silva, William H. Warren, and Michael J. Tarr. Comparing 2d vector field visualization methods: A user study. *IEEE Transactions on Visualization and Computer Graphics*, 11:59–70, 2005.

[NJP05] Alisa Neeman, Boris Jeremic, and Alex Pang. Visualizing tensor fields in geomechanics. *Visualization Conference, IEEE*, 0:5, 2005.

[NLK+03] A. M. Nayak, K. Lindquist, D. Kilb, R. Newman, F. Vernon, J. Leigh, A. Johnson, and L. Renambot. Using 3D Glyph Visualization to Explore Real-time Seismic Data on Immersive and High-resolution Display Systems. *AGU Fall Meeting Abstracts*, pages C1208+, December 2003.

[ODM+06] K. B. Olsen, S. M. Day, J. B. Minster, Y. Cui, A. Chourasia, D. Okaya, P. Maechling, and T. Jordan. Competition data set and description in 2006 ieee visualization design contest, 2006.

[ODM+08] K. B. Olsen, S. M. Day, J. B. Minster, Y. Cui, A. Chourasia, D. Okaya, P. Maechling, and T. Jordan. TeraShake2: Spontaneous Rupture Simulations of Mw 7.7 Earthquakes on the Southern San Andreas Fault. *Bulletin of the Seismological Society of America*, 98(3):1162–1185, 2008.

[Sha05] Nima Bigdely Shamlo. Matlab toolbox for high resolution vector field visualization with application in improving the understanding of crack propogation mechanisms. Master's thesis, San Diego State University, 2005.

[SWBG06] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based ray-casting of quadratic surfaces. In *Eurographics Symposium on Point-Based Graphics*, pages 59–65. Citeseer, 2006.

[TYRg+06] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-guzman, Jacobo Bielak, Omar Ghattas, Kwan liu Ma, and David R. O'hallaron. From mesh generation to scientific visualization: An end-to-end approach. In *in SC2006*, 2006.

[WMM+02] C.-F. Westin, S. E. Maier, H. Mamata, A. Nabavi, F. A. Jolesz, and R. Kikinis. Processing and visualization of diffusion tensor MRI. *Medical Image Analysis*, 6(2):93–108, 2002.

[YMW04] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. A parallel visualization pipeline for terascale earthquake simulations. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 49, Washington, DC, USA, 2004. IEEE Computer Society.