University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, & Professional Papers

Graduate School

2005

# Transparent Line Integral Convolution: A new approach for visualizing vector fields in OpenDX

Alexander Petrov Petkov
*The University of Montana*

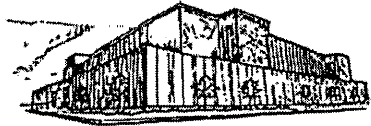
Follow this and additional works at: https://scholarworks.umt.edu/etd

Let us know how access to this document benefits you.

Recommended Citation

Petkov, Alexander Petrov, "Transparent Line Integral Convolution: A new approach for visualizing vector fields in OpenDX" (2005). *Graduate Student Theses, Dissertations, & Professional Papers*. 5077.
https://scholarworks.umt.edu/etd/5077

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike
MANSFIELD LIBRARY

The University of

**Montana**

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

**Please check "Yes" or "No" and provide signature**

Yes, I grant permission ✓

No, I do not grant permission ______

Author's Signature: [signature]

Date: May 5, 2005

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

8/98

**Transparent Line Integral Convolution:**

**a new approach for visualizing vector fields in OpenDX**

by

Alexander Petrov Petkov

B.M., The University of Arizona, 1996

presented in partial fulfillment of the requirements

for the degree of

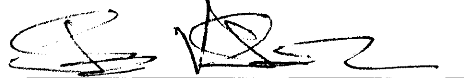
Master Of Science

The University of Montana

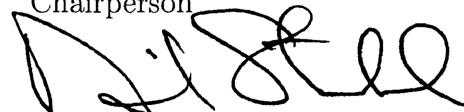
Missoula, Montana

April, 2005

Approved by:



Chairperson



Dean, Graduate School

5-5-05

Date

UMI Number: EP40541

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI EP40541

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

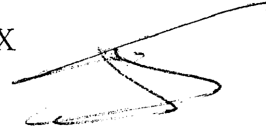
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Pctkov, Alexander P, M.S., April, 2005 Computer Science

Transparent Line Integral Convolution:

a new approach for visualizing vector fields in OpenDX



Director: Dr. Jesse V. Johnson

## ABSTRACT

Traditional techniques for visualizing vector fields consist of using glyphs, streamlines, or streaklines. The Line Integral Convolution method (LIC) is examined as an alternative approach for vector field visualization. This method is based on integrating a texture along computed flow lines, and creates a continuous texture representation of the vector field. LIC eliminates some visualization difficulties, such as vector glyphs using too much of the display and obscuring other elements of interest, and has proven useful for representing large vector fields.

This thesis is focused on developing a LIC module for the OpenDX (http://www.opendx.org) environment. The original algorithm is extended through the use of transparency and animation. The module can create texturized vector field flow, which can be studied simultaneously with other data elements, produced by the visualization environment.

LIC is demonstrated here to provide a superior visualization alternative for both "real world" and idealized data sets. The result of this thesis will benefit researchers from various disciplines, who will be able to use the LIC module within OpenDX for the visualization of large vector fields as continuous texture maps. Possible applications include the modeling of weather systems, computational fluid dynamics, electromagnetic fields, and ice sheets on Mars.

The LIC module for OpenDX will be released to the open source community.

## ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis.

First and foremost, Dr. Jesse Johnson for his guidance throughout this research and the writing of this thesis. Moreover, he has continuously inspired me during the course of my graduate education.

I would also like to thank my committee members for their efforts and contributions to this work: Dr. Ray Ford and Dr. Andrew Ware.

I thank the employees at *Vizsolutions Inc.* for their continuous OpenDX development, my friend and colleague Jared Rapp for his constructive criticism, and my wife, Phyllis for her ongoing support throughout the course of my graduate studies.

# TABLE OF CONTENTS


**ABSTRACT** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ii

**ACKNOWLEDGMENTS** . . . . . . . . . . . . . . . . . . . . . . . . iii

**CHAPTER 1 INTRODUCTION** . . . . . . . . . . . . . . . . . . . . 1

Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

Benefits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . 5

**CHAPTER 2 OVERVIEW** . . . . . . . . . . . . . . . . . . . . . . 6

Related Literature . . . . . . . . . . . . . . . . . . . . . . . . . 6

Icons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Streamlines/Streaklines . . . . . . . . . . . . . . . . . . . . . 6

Spot-noise Algorithm . . . . . . . . . . . . . . . . . . . . . . . 7

Line Integral Convolution . . . . . . . . . . . . . . . . . . . . . 9

What is a Vector? . . . . . . . . . . . . . . . . . . . . . . . . . . 10

IceView . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

Vector Field Visualization in IceView . . . . . . . . . . . . . . . 13

Data Explorer (OpenDX) . . . . . . . . . . . . . . . . . . . . . . . 13

Execution Model . . . . . . . . . . 15
Visual Programming Environment . . . . . . . . . . 16
Application Program Interface . . . . . . . . . . 17
Data Model . . . . . . . . . . 19
**CHAPTER 3 METHODS** . . . . . . . . . . 21
Line Integral Convolution . . . . . . . . . . 21
Random Noise Texture . . . . . . . . . . 25
Computing the Streamline . . . . . . . . . . 26
Computing the Weight $h_i$ . . . . . . . . . . 26
Kernel Function . . . . . . . . . . 27
Computing Output Pixel Value . . . . . . . . . . 30
Euler Method . . . . . . . . . . 30
Runge-Kutta Method . . . . . . . . . . 31
Numerical Methods Evaluation . . . . . . . . . . 33
Alpha Blending . . . . . . . . . . 34
**CHAPTER 4 IMPLEMENTATION AND RESULTS** . . . . . . . . . . 36
Implementation Plan . . . . . . . . . . 36
Initial Prototype . . . . . . . . . . 36
Interface Design . . . . . . . . . . 36
Implementation . . . . . . . . . . 37
Test Suite . . . . . . . . . . 38
Results . . . . . . . . . . 39
User Control . . . . . . . . . . 42
Alpha Blending . . . . . . . . . . 42
Animation . . . . . . . . . . 42

Invalid Positions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**CHAPTER 5 CONCLUSIONS AND FUTURE DIRECTIONS** . . 46

Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Known Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

Future Directions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**BIBLIOGRAPHY** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

# LIST OF TABLES

Table 2.1 Module types in OpenDX and their descriptions. . . . . . . . 18

Table 2.2 Array types in OpenDX and their descriptions. . . . . . 20

Tablc 3.1 Accumulated error and number of function evaluations comparison for the numerical methods. . . . . . . . . . . . . . . . . 34

Table 4.1 Interface design for the OpenDX LIC module. . . . . . 37

Table 4.2 Test suite for the OpenDX LIC module. . . . . . . . . . 39

# LIST OF FIGURES


Figure 1.1 The velocity of the ice field for a large dataset after resampling. Although the underlying topography is easier to see, the number of velocity points has been reduced, perhaps missing some information of interest. . . . . . . . . . . . . . . . . . . . . . 4

Figure 2.1 Icon-based representation for a vector field. . . . . . . . . 7

Figure 2.2 Streamline representation for a vector field. . . . . . . . . . 8

Figure 2.3 The result from the spot-noise algorithm. Figure in [17]. . . . 9

Figure 2.4 LIC-based representation of circular and turbulent fluid dynamics vector fields. Figure in [2]. . . . . . . . . . . . . . . . . 10

Figure 2.5 IceView output, showing ice sheet and its velocity, continental bed topography and temperature, and basal water conditions. 12

Figure 2.6 IceView output, showing velocity of the ice field for the same time frame as in Figure 1.1 without resampling. The large number of vectors obscures the topography. . . . . . . . . . . 14

Figure 2.7 The OpenDX execution hierarchy. Programs can be DX scripts, standalone applications, as well as graphical user interfaces that control the DX executive. . . . . . . . . . . . . . . . . . . 15

Figure 2.8 The OpenDX visual programming environment. Modules are "wired" into one another to create a program yielding visual results. . . . . . . . . . . . . . . . . . . . . . . . . . 16

Figure 2.9 Using the OpenDX API: developing new modules, controlling DX from a GUI, or writing a standalone application. Figure in [4]. . . . . . . . . . . . . . . . . . . . . . . . . . 17

Figure 3.1 The LIC algorithm operates on a vector field and a noise texture. The result is a textured pattern for the flow of the vector field. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

Figure 3.2 For each pixel in the input noise texture... . . . . . . . . . . . 23

Figure 3.3 ...compute the streamline for user-specified length $l$ in positive and negative direction. . . . . . . . . . . . . . . . . . . . . 23

Figure 3.4 For each point in the streamline, compute the weight $h_i$. . . . 24

Figure 3.5 Compute the output pixel value by using the input pixel value and the computed weights in Figure 3.4. . . . . . . . . . . . . 24

Figure 3.6 A 2-dimensional array with randomly generated numbers. The array has the same size as the vector field and will serve as a white-noise texture input to the LIC module. . . . . . . . . . 25

Figure 3.7 A computer-generated graphic of a tree before (left) and after applying Gaussian blur. The filtered version appears to have less detail, especially around the edges of the tree. Image courtesy of Jared D. Rapp, University of Montana. . . . . . 28

Figure 3.8 Phase-shifted Hanning ripple functions(top), a Hanning windowing function(middle), and their product (bottom). Figure from [2]. . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

Figure 3.9 Accuracy for the numerical methods. The two Runge-Kutta variants are closer than the Euler method to the exact solution. 33

Figure 3.10 An example of alpha blending with alpha values of 0, 0.5, and 1 respectively. Image courtesy of Duane Bong, visionengineer.com. 34

Figure 4.1 Inputs to the LIC module can be specified, as well as routed from other modules. . . . . . . . . . . . . . . . . . . . . . . . 38

Figure 4.2 Gwenn Flowers' glacial flood data. Shown are glyphs (top), where arrows are scaled to velocity magnitude, streamlines (middle), and the texturized LIC flow (bottom). . . . . . . . 40

Figure 4.3 Matlab Peaks: comparing glyphs (top), streamlines (middle), and LIC (bottom). . . . . . . . . . . . . . . . . . . . . . . 41

Figure 4.4 An electric field, visualized with the LIC module. The two images differ as a result of altering the streamline length and kernel function parameters. . . . . . . . . . . . . . . . . . . . . 42

Figure 4.5 The result from the LIC module for OpenDX , superimposed over the magnitude of the vector field. The added opacity level enables the viewer to see the color-coded magnitude, as well as the flow of the vector field. . . . . . . . . . . . . . . . . . 43

Figure 4.6 Texturized velocity flow of an ice field, superimposed over the topography of North America and Greenland, before and after invalidating positions with zero vector values. . . . . . . . . . 45

# CHAPTER 1 INTRODUCTION

## Introduction

In our lives we have witnessed rapid advancements in the development of computer storage, processing and graphics technologies. These developments have enabled the scientific community to store data in larger quantities, to process it faster, and to present findings by using data visualization:

> *"...data visualization becomes more and more widespread in science, both because today's computer hardware make it easy to produce pictures, and because pictures have inherent power to convey complex information..."* [10]

Data visualization has proven to be an invaluable way of sharing knowledge and ideas. Humans have the ability to analyze vast quantities of visual information very quickly. Data visualization uses this ability by engaging what is arguably the most sophisticated sensory perception [19][1].

The benefits from data visualization are numerous. For example, the visual representation of data is very useful in presentation setting, as it helps to communicate complex ideas quickly and effectively [12]. Data visualization is also crucial for data

[1] *A Model of Perceptual Processing*, pp. 25-27 in the text.

analysis, where overall data patterns are easily recognized, as well as identifying inconsistencies in the data. Furthermore, visualization is often the basis of forming hypotheses for future research [19][2].

Software packages for data visualization are being actively developed and constantly improved. One such software package is OpenDX[3]—originally from IBM, and later released as open source software. The University of Montana takes an active part in furthering OpenDX development, thanks to the efforts of Dr. Ray Ford, David Thompson and Jeff Braun at *Vizsolutions, Inc.*

Very often we see the need to visually represent vector fields, since vectors are frequently used to describe motion. Traditional vector field visualization techniques are restricted to icon-based symbols [16], and line (streamline, or streakline) representation [6]. In the case of dynamical systems (*e.g.,* fluid flow, magnetic fields), these common approaches are often inadequate. They can provide only a rough overview of the underlying dynamics, and often produce cluttered and confusing images [11]. To address these inadequacies, a number of new, texture-based methods to visualize vector fields have been developed—originating with van Wijk's spot-noise algorithm [17], and later followed by Cabral and Leedom's Line Integral Convolution method [2].

## Motivation

The idea for an alternative approach to vector field visualization originated from IceView development. IceView[4] is an interactive program that models glaciation (ice cover and other glaciation elements), developed for the OpenDX visualization

[2]See p. 2 in the text.
[3]See http://www.opendx.org
[4]A detailed IceView commentary follows in Chapter 2

environment.

During development, it was observed that the ice velocity field for large datasets can be very dense. The traditional use of glyph icons for the velocity lead to crowding the display, which presented difficulties for the viewer to extract visual information. In addition, underlying data elements were obstructed by the ice velocity field.

It became evident that compromises need to be made. On a large scale (*e.g.*, viewing North America), the vector field needs to be resampled so the user can view overall velocity patterns (*e.g.*, direction). One of the current resampling techniques used in OpenDX is to reduce the number of vectors displayed on the screen (Figure 1.1). This is an imperfect solution, since information of interest (*e.g.*, fast moving ice formations) may be discarded as the result of resampling. Moreover, it is difficult for the viewer to reconstruct the animated ice flow from discrete sample points while visualizing time series data.

With respect to this problem, the Line Integral Convolution (LIC) method has been studied as a texture-based alternative, capable of displaying large vector fields without resampling. A complete analysis of the LIC method is given in Chapter 3.

## Goal

Studying the LIC method lead to the idea of visualizing vector fields as a continuous texture, such that overall patterns in the field are not lost as a result of resampling, as well as to allow for the visual presence of other dataset elements by using alpha-blending (transparency).

Therefore, the work in this thesis is focused on extending the LIC method to use transparency. This concept is implemented as an OpenDX module, providing a texture-based alternative for visualizing vector fields in OpenDX.

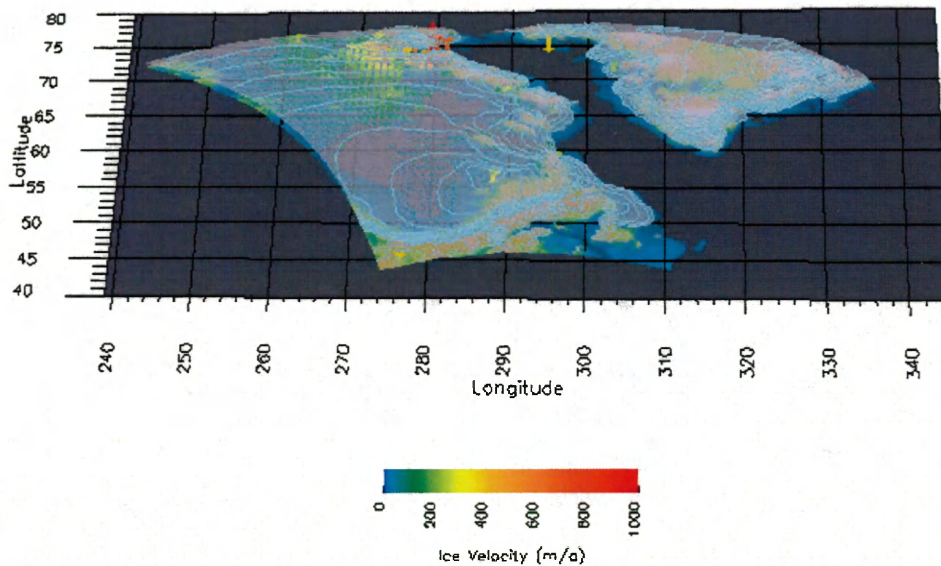



Figure 1.1 The velocity of the ice field for a large dataset after resampling. Although the underlying topography is easier to see, the number of velocity points has been reduced, perhaps missing some information of interest.

## Benefits

Given OpenDX's capabilities to overlay a multitude of elements, this implementation is expected to prove beneficial to anyone using the OpenDX programming environment for visualizing dense vector fields. Practical applications include the modeling of weather systems (such as hurricanes and cloud formations), magnetic fields, as well as any other field where each data point is associated with more than one value.

## Thesis Organization

The rest of this thesis is organized as follows:

- **Chapter 2** provides an overview of related literature, key concepts, and software elements used in this research.
- **Chapter 3** addresses the methods used for developing the OpenDX LIC module.
- **Chapter 4** describes the implementation, test data sets, and results.
- **Chapter 5** contains conclusion remarks, as well as an outline for future work.

# CHAPTER 2 OVERVIEW

## Related Literature

### Icons

Perhaps the most popular approach to vector field visualization is by using icons ([16], [19][1]). These icons are most often in the form of arrows, and are generated for each data point in the vector field. This approach can be very effective: the length of the arrows is indicative of the magnitude, and their orientation shows direction (Figure 2.1).

In the case of dense vector fields, however, the display becomes cluttered, since an icon is generated for each data point (Figure 2.6). A solution is to reduce the number of icons on the screen by sampling the data set (*e.g.,* , consider every third point). This is an imperfect solution, since potentially important data is not visualized.

### Streamlines/Streaklines

Another traditional technique for vector field visualization is the drawing of curves. These curves can be tangential to the vector field (streamlines, Figure 2.2), or they can be line traces of particles in a changing vector field through time (streaklines, [4]).

---

[1] Perceiving direction: Representing Vector Fields, p. 216.

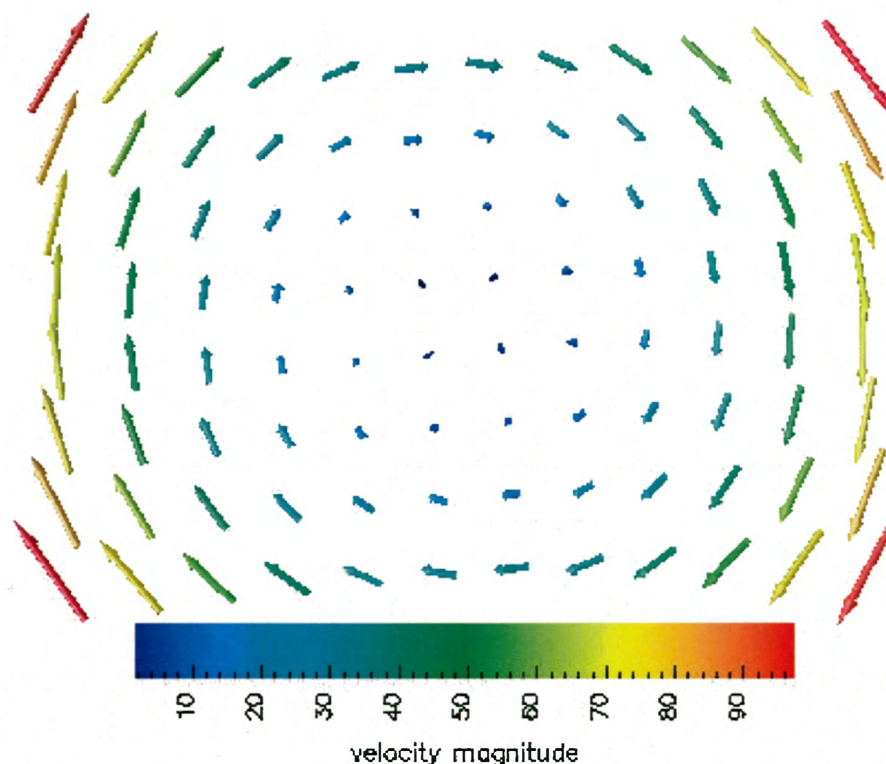



Figure 2.1 Icon-based representation for a vector field.

Although streamlines can produce a coherent image of the flow pattern, the sense of direction is lost [19][2]. Moreover, streamline computation depends on placement of arbitrary "seed points" [6], which can potentially lead to loss of subtle trends in the data.

### Spot-noise Algorithm

Van Wijk [17] originated a technique for texture-based vector field representation. The author uses a spot noise texture, which consists of randomly inserted "spots" of arbitrary shape (*e.g.,* squares, ellipses) and random intensity [17]. The spot noise

[2]Perceiving direction: Representing Vector Fields, p. 217.

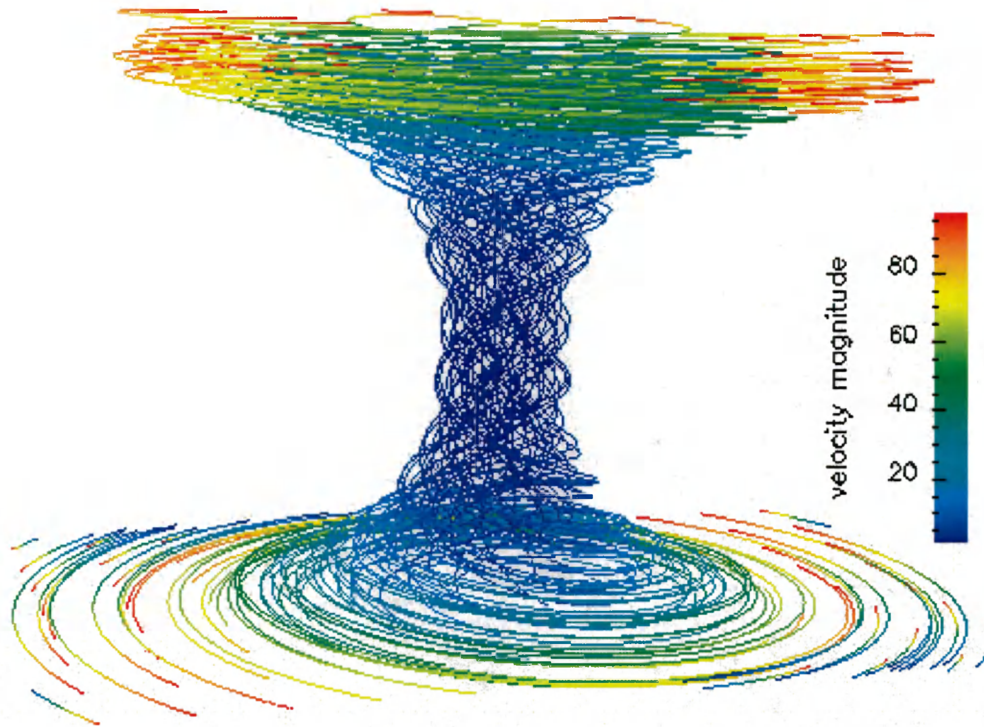



Figure 2.2 Streamline representation for a vector field.

texture is convoluted to a straight line segment, parallel to the direction of each vector (Figure 2.3).

This method depicts all parts of the vector field without competing for display resolution. However, it is better suited for a particular class of vector data [2]. In particular, details of highly-curved vector field flow may be lost as a result of the straight line approximation of the local vector field, as well as the choice for the spot shape in the noise texture.

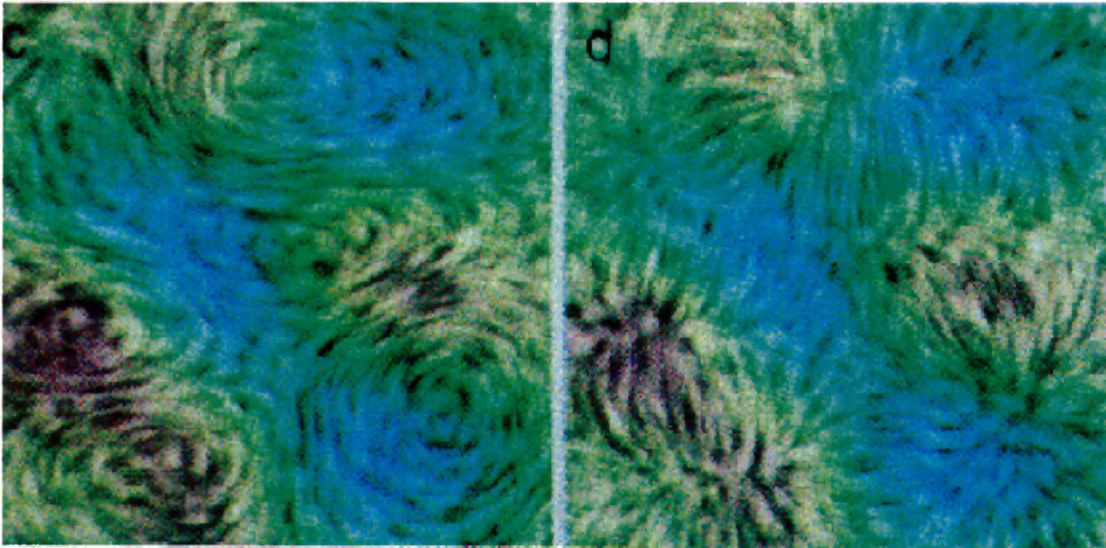



Figure 2.3 The result from the spot-noise algorithm. Figure in [17].

### Line Integral Convolution

The Line Integral Convolution method (LIC, [2]) was originally authored by Cabral and Leedom in 1993. It is known as a modern and highly effective texture-based technique for visualizing dense vector fields, where the texture is an image with pixel colors generated at random.[3] With the help of an advection method, the result is an image, showing the texturized flow of the vector field (Figure 2.4).

Unlike the spot-noise algorithm, LIC computes line segments which are tangential to the flow of the vector field. As a result, the technique produces striking images, capable of revealing intricacies of the vector field flow. A drawback for LIC is that orientation cannot be perceived from a single image. For example, in the case of circular flow, it cannot be observed if the direction is clockwise or counter clockwise. A way to overcome this limitation is outlined further in Chapter 3.

[3]The texture is explained on p. 25

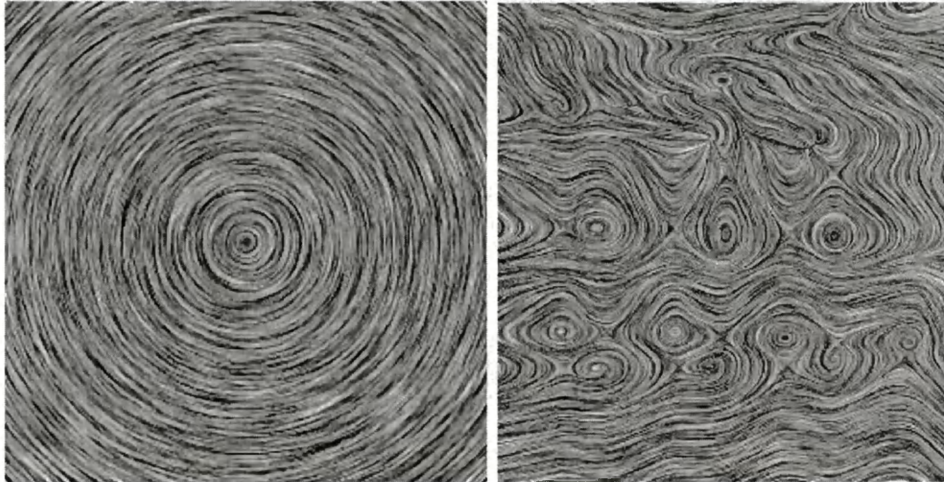


Figure 2.4 LIC-based representation of circular and turbulent fluid dynamics vector fields. Figure in [2].

## What is a Vector?

In one of its simplest forms, a vector is used to describe the position and motion of a particle in a 2-dimensional plane. For our purposes, the following vector definition is used[4]:

> *"A vector is an entity that is specified by a magnitude and direction."* [8]

In particular, if a particle has a position described by $x(t)$ and $y(t)$ at time $t$, then the vector $\vec{v} = (v_x, v_y)$ shows the displacement per unit time of the particle at time $t$, and can be used to describe the speed (the magnitude $\hat{v}$ of vector $\vec{v}$) and the direction of that particle.

As an example, consider the velocity vector, which is the rate of change in the position of a particle, given that the position is a known function of time:

[4]See p. 549 in the text.

$$v_x = \frac{dx(t)}{dt}, v_y = \frac{dy(t)}{dt} \tag{2.1}$$

Once the vector elements are known, we can use the Pythagorean theorem and trigonometric identities to find magnitude and direction. More specifically, the magnitude of a vector $\vec{v} = (v_x, v_y)$ is given by:

$$|v| = \sqrt{v_x^2 + v_y^2} \tag{2.2}$$

and the direction can be found by calculating the angle $\theta$:

$$\theta = \tan^{-1} \frac{v_y}{v_x} \tag{2.3}$$

## IceView

IceView is an interactive visualization program that models glaciation (ice cover), and is developed in the OpenDX visual programming environment.

IceView was started based on demand to increase the understanding of ice sheet dynamics over the last ice age, with relation to climatic and geological factors. As an extension to the University of Maine Ice Sheet Model (UMISM), IceView provides highly informative 3-dimensional ice flow visualizations as time series movies at various scales and resolutions. Development efforts have resulted in the following modeling features (Figure 2.5):

- Thickness of the ice cover
- Growth and decay of ice caps and ice sheets
- Velocity vectors and flow lines

- Temperature of the bed under the ice
- Bed topography during and after glaciation
- Wet and dry glacier bed conditions
- Other user-derived quantities

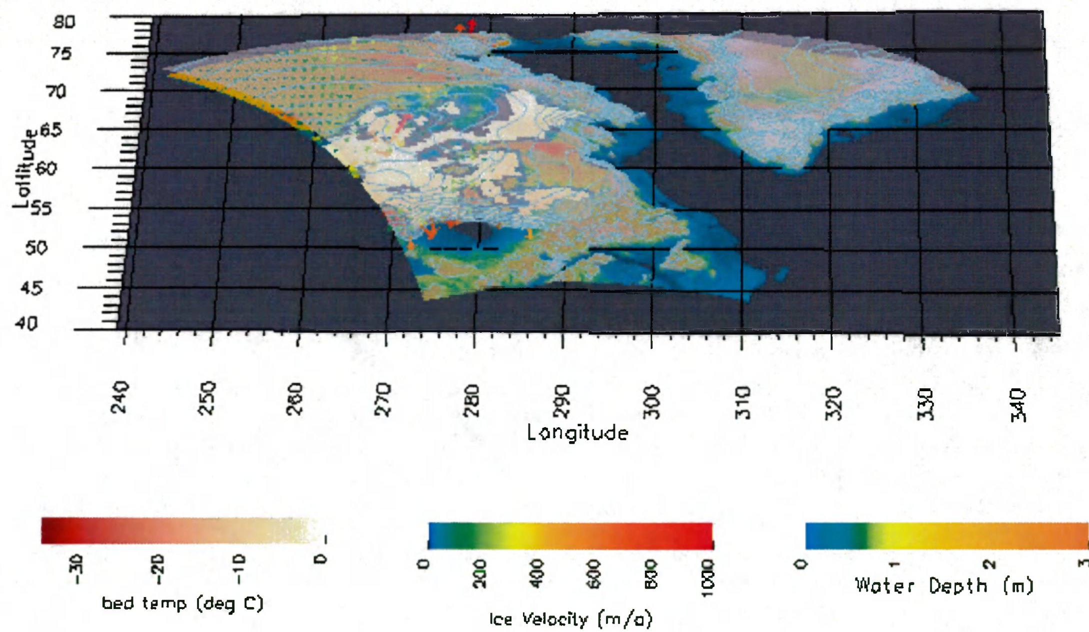



Figure 2.5 IceView output, showing ice sheet and its velocity, continental bed topography and temperature, and basal water conditions.

### Vector Field Visualization in IceView

Currently, ice velocity is shown with the help of traditional vector field visualization techniques, therefore restricted to glyph[5], streamline, or streakline representation [15]. On a large scale (*e.g.,* viewing North America), these common approaches are often inadequate—the vector field needs to be resampled so the user can view overall velocity patterns, such as the direction of the velocity. The current resampling technique used in IceView is achieved by allowing the user to reduce the number of vectors displayed on the screen (Figure 2.6). This is a less desirable solution, since information such as fast moving ice formations may be discarded as the result of resampling, and the resampling scheme often fails at the boundaries.

## Data Explorer (OpenDX)

Data Explorer from IBM (OpenDX ) is a powerful and flexible software package, utilized by users of all levels (programmers and novices alike) to visualize and analyze data. This is accomplished through the many sample programs available, as well as the capability for others to write extensions (modules) for OpenDX. Moreover, OpenDX can visualize data from all areas of knowledge—medicine, geology, mechanics, and more. It is concisely described as follows:

> *"Data Explorer is a visualization system that can be used in many application areas and with a variety of data representations to extract useful information from complex data."* [4]

[5] A few examples of glyph icons are arrows, needles, or spheres

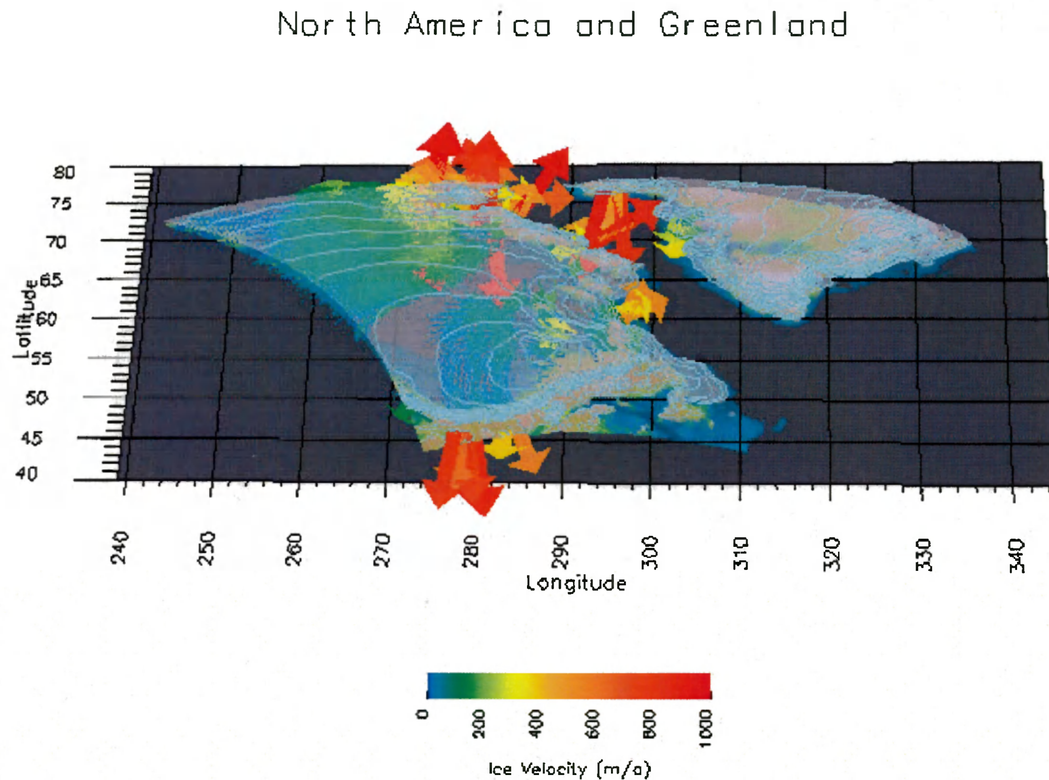



Figure 2.6 IceView output, showing velocity of the ice field for the same time frame as in Figure 1.1 without resampling. The large number of vectors obscures the topography.

OpenDX contains a large set of visualization tools in the form of modules. A module can be accessed and used in a variety of ways. For example, a module can be used as [4]:

- a node through the use of its icon in a visual programming network
- a function call, available in the scripting language interface provided by the executive layer

- a part of the OpenDX API

The remaining material in this chapter is a descriptive summary of OpenDX components, which are explained in greater detail in [4].

### Execution Model

OpenDX has a client-server execution model, which is well suited for single and multi- processor machines. As demonstrated in Figure 2.7, clients (programs) can connect to a server in a variety of ways:

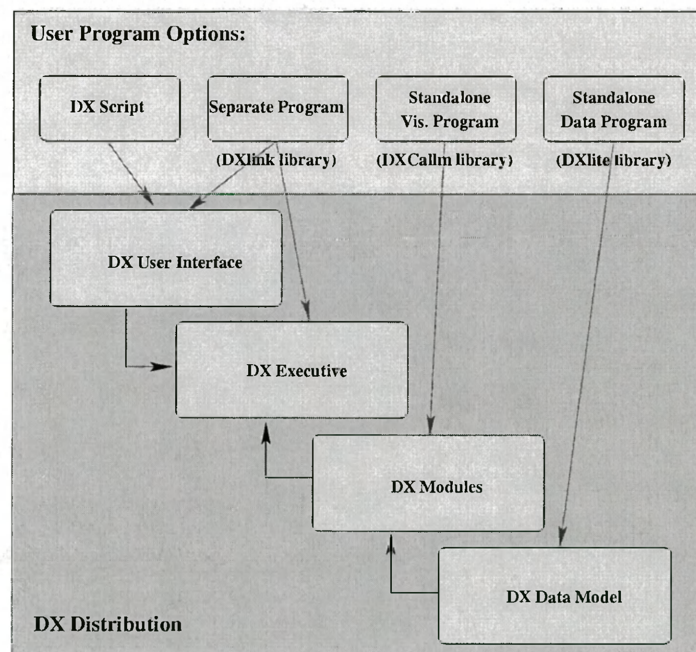



Figure 2.7 The OpenDX execution hierarchy. Programs can be DX scripts, standalone applications, as well as graphical user interfaces that control the DX executive.

## Visual Programming Environment

OpenDX's Visual Programming Environment (VPE) helps users to easily create programs via the point-and-click interface. Users can select any available operations (modules), and place them on the canvas. Different modules are connected (or wired) together, thus dictating the logic for the program execution (Figure 2.8):

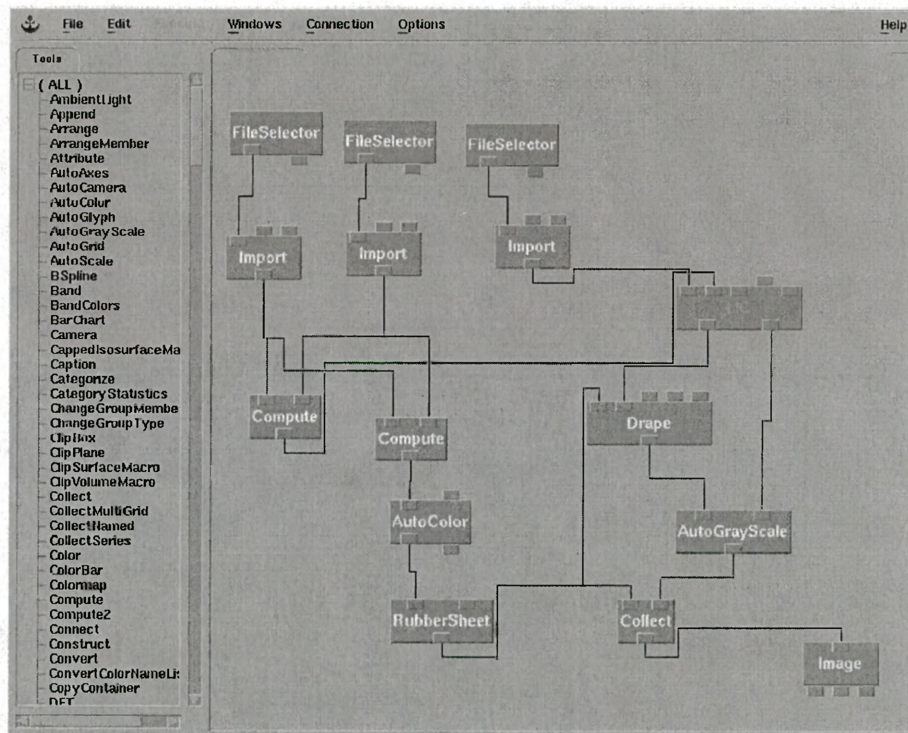



Figure 2.8 The OpenDX visual programming environment. Modules are "wired" into one another to create a program yielding visual results.

A number of modules in OpenDX allow for program execution control and user interaction via intuitive interfaces (*e.g.,* the sequencer module, which functions like a VCR). Advanced users seeking more control can also create programs by using the scripting language capabilities in OpenDX .

### Application Program Interface

OpenDX's well-documented application program interface (API) facilitates the development of stand-alone user programs, as well as additional OpenDX functions (modules). Figure 2.9 shows the options for using the API:

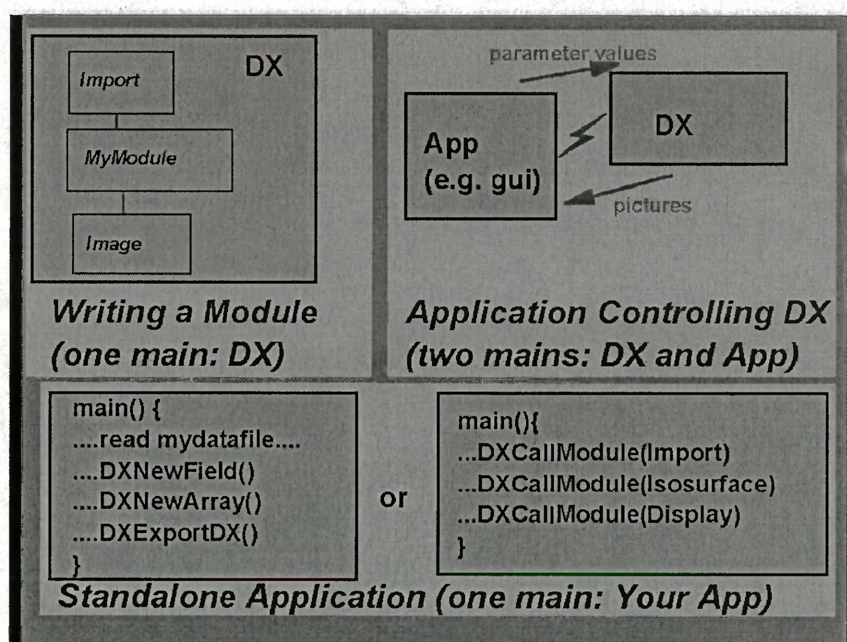



Figure 2.9 Using the OpenDX API: developing new modules, controlling DX from a GUI, or writing a standalone application. Figure in [4].

New modules can be either inboard, outboard, or a runtime-loadable [3][6]. Table

[6]Section 11.3, as of Feb, 2005

2.1 shows their differences:

| Module Type | Description |
|---|---|
| inboard | • requires a separate version of DX executive<br>• efficient<br>• runs as a single process |
| outboard | • separate user module executive<br>• runs as a separate process<br>• less efficient |
| run-time loadable | • separate user module executive<br>• linked in at runtime<br>• efficient<br>• runs as a single process |

Table 2.1 Module types in OpenDX and their descriptions.

The **DX module builder** facilitates the writing of additional modules by presenting users with a graphical user interface for specifying inputs and outputs for a new module, as well as placing the new component into an appropriate category (transformation, realization, etc.). This interface can also build a C-code framework file, a module description file, and a makefile for compiling the new module.

### Data Model

OpenDX has the ability to import and use data from a variety of formats: binary, NetCDF, HDF, spreadsheet, ASCII, and more. The software also has its native data model, which facilitates the process of describing data to a great extent. This data model supports various types of simulation and observational data, and can represent a variety of data structures [4]:

- Data on a regular orthogonal grid.
- Data on a deformed regular or curvilinear grid.
- Data on irregular grids.
- Unstructured data with no regular connection between the data samples.

#### Object Types

Data are stored as Objects, which are used by OpenDX modules:

> *"An object is a data structure stored in memory, that contains an indication of the Object's type, as well as other time-dependent information."* [4]

In practice, much of the data is represented by Array objects. What follows is a brief listing of the most common object types in OpenDX.

1. Fields

   Field objects are the constitutive part in the OpenDX data model. A field represents a set of data that is associated with positions and (usually) connections.

Thus, the data, positions, and connections are "components" of a field. The data model allows for sharing of the same components between different fields.

2. Groups

   The group objects are compound objects, used to collect members members that themselves may be fields and/or groups. A group object is often used to collect series (*e.g.,* time series). It cannot collect components, where the field object is most suitable. Each group member may be referenced either by name or index.

3. Arrays

   Array objects in OpenDX can hold the actual data, positions, connections, and other field components. Table 2.2 lists the types of arrays in OpenDX, as described in [4].

| **Array Type** | **Description** |
|---|---|
| Regular Array | One-dimensional series of evenly spaced points |
| Irregular Array | A general way to specify the contents of an array by simply listing the values |
| Path Array | One-dimensional series of connected line segments |
| Product Array | Regular or semi-regular grid positions |
| Mesh Array | Regular or semi-regular grid connections |
| Constant Array | Array with a constant value |

Table 2.2 Array types in OpenDX and their descriptions.

4. Attributes

   An attribute defines the association between an OpenDX object (array, component, field, or group) and a value (simple or compound). Typically, an attribute associates an object with a data segment.

# CHAPTER 3 METHODS

## Line Integral Convolution

The LIC algorithm is a way to represent a 2 or more dimensional vector field as a continuous map. This is accomplished by using an image (noise texture) with randomly-generated pixel colors and a vector field as inputs to the LIC method. Both inputs have the same size (Figure 3.1).

The input texture is filtered along computed, curved streamline segments. LIC uses a one-dimensional low-pass filter[1] to convolute the input noise texture by following the direction of the flow lines in the vector field [13].

The following pseudocode describes how the LIC algorithm works:

1. For each pixel in the input noise texture (Figure 3.2):
   (a) Compute the streamline for user-specified length $l$ in positive and negative direction (Figure 3.3).
   (b) For each point in the streamline, compute its convolution weight $h_i$ (Figure 3.4).
   (c) Compute the output pixel value by using the input pixel values and the computed weights in (b) (Figure 3.5).

[1]Defined on p. 27

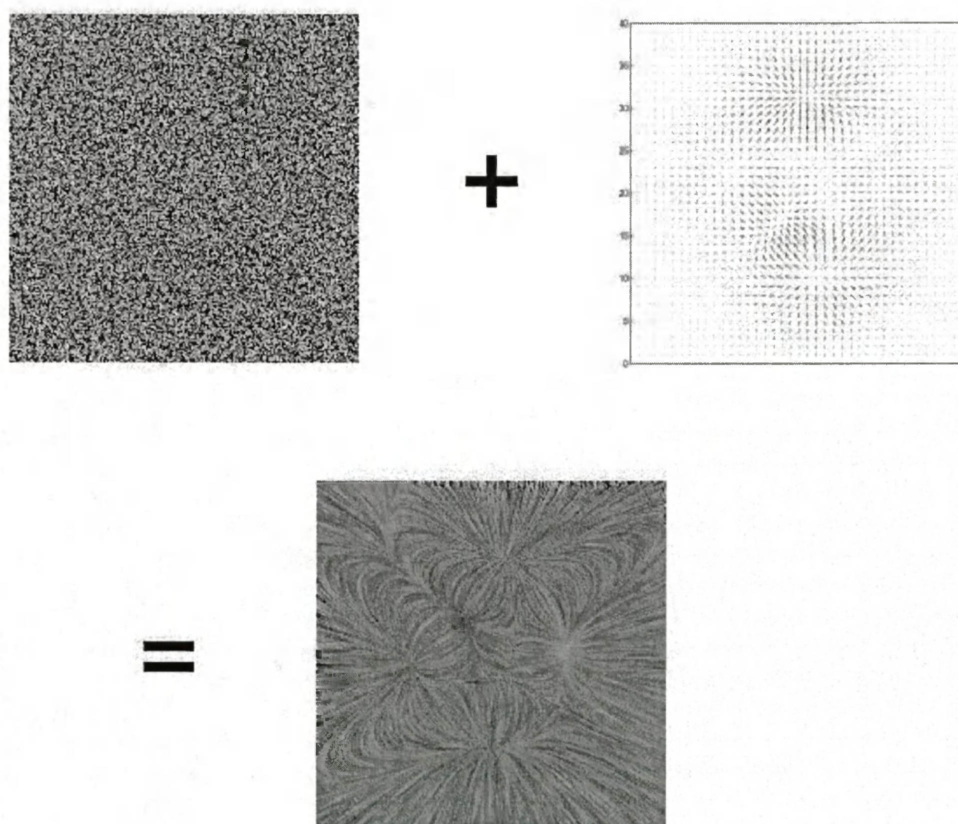


Figure 3.1 The LIC algorithm operates on a vector field and a noise texture. The result is a textured pattern for the flow of the vector field.

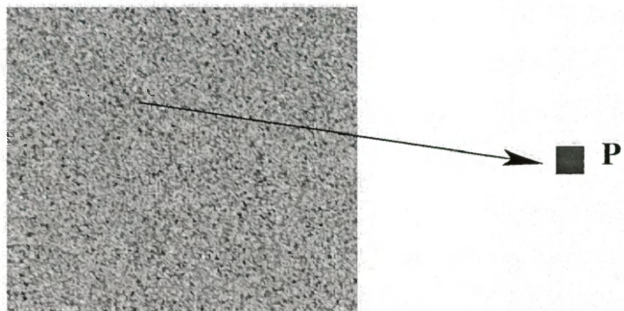



Figure 3.2 For each pixel in the input noise texture...

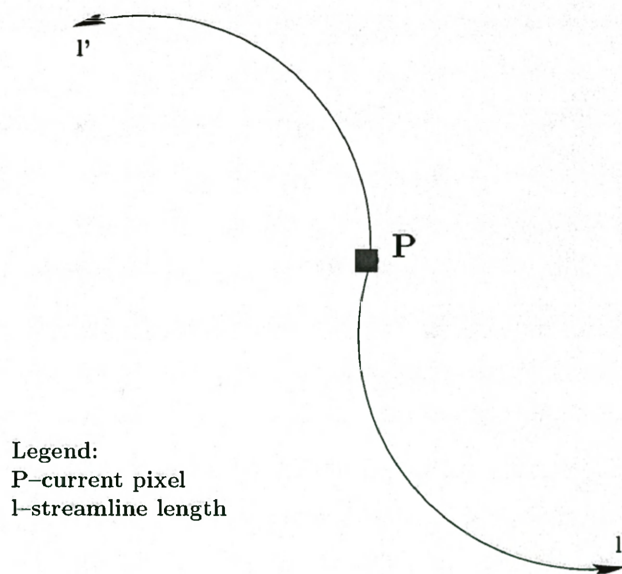



Figure 3.3 ...compute the streamline for user-specified length $l$ in positive and negative direction.

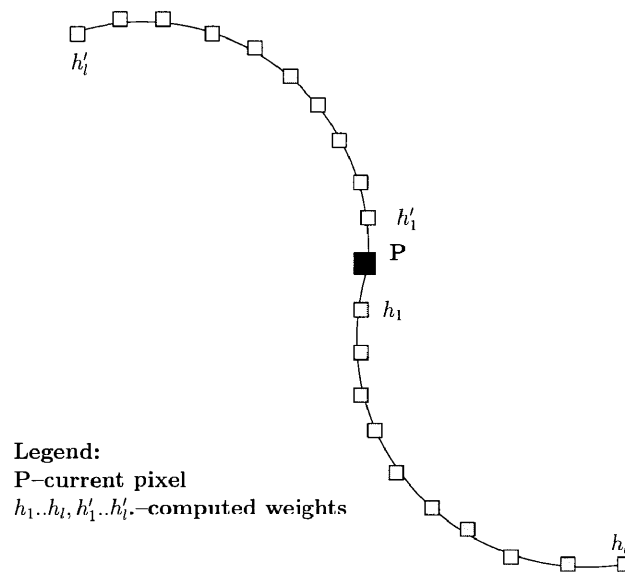



Figure 3.4 For each point in the streamline, compute the weight $h_i$.

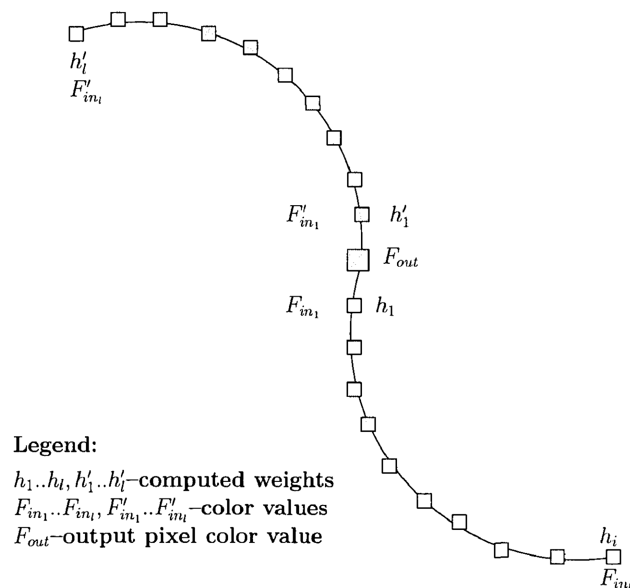



Figure 3.5 Compute the output pixel value by using the input pixel value and the computed weights in Figure 3.4.

### Random Noise Texture

The random noise texture is constructed by simply generating a 2-dimensional array of random numbers. It is important to emphasize that this generated array is of the same size as the input vector field, such that each pixel has a corresponding vector element. The random number values are between a desired range (often between 0 and 255, standard in graphics packages), and represent the pixels in the input noise texture:

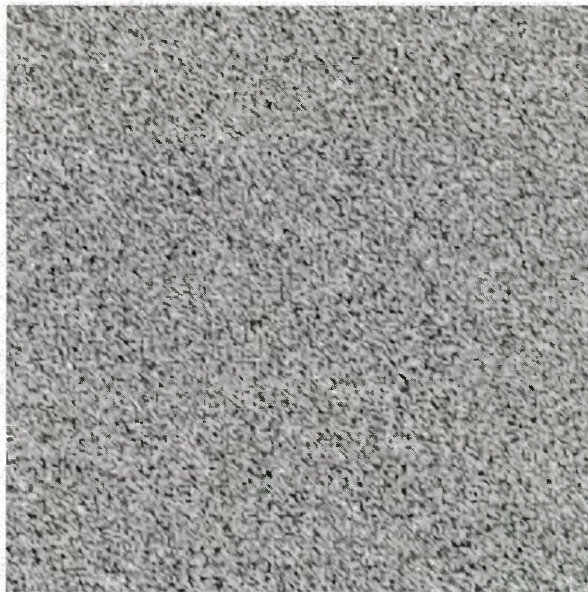


Figure 3.6 A 2-dimensional array with randomly generated numbers. The array has the same size as the vector field and will serve as a white-noise texture input to the LIC module.

### Computing the Streamline

Computing the streamline for each pixel can be done by using numerical-integration schemes such as the Runge-Kutta variants. The authors use the adaptive step Euler method for computing the streamline, where each consecutive point in the streamline is found by using the position for the previously-found point and its velocity:

$$P_0 = (x + 0.5, y + 0.5) \tag{3.1}$$

$$P_i = P_{i-1} + \frac{V(P_{i-1})}{||V(P_{i-1})||}\Delta s_i \tag{3.2}$$

where $P_0$ is the pixel for which we want to compute the streamline, $P_i$ is the next point in the streamline, $V(P_{i-1})$ is the vector value at the previous point in the streamline, $\Delta s_i$ is the current step size[2], and $i$ goes from 1 to $l$, the desired streamline length (Figure 3.3).

### Computing the Weight $h_i$

Computing the weight $h_i$ for each point in the streamline is done by finding the exact integral of the convolution kernel $k$[3] at every numerical integration step in the streamline:

$$h_i = \int_a^b k(\omega)d\omega \tag{3.3}$$

where the index $i$ represent the index of the current point in the streamline, $a$ is the distance along the streamline from the point for which we want to compute the

[2]At this time the LIC module uses $\Delta s_i = 1$

[3]Defined later on p. 28

output value, and $b$ equals $a$ plus the current step size $\Delta s_i$ from Equation 3.1 (Figure 3.4).

### Kernel Function

From Figure 3.1, it is evident that the result of the LIC algorithm shows details of the vector field flow very well, while the sense of direction is lost. To overcome this limitation, Cabral and Leedom use a periodic phase shift filter, known as the Hanning function [2]:

$$\frac{1}{2}\left[1 + cos(d\omega + \beta)\right] \tag{3.4}$$

where $d$ is a dilation constant and $\beta$ is the phase shift value, given in radians.

The Hanning function has properties of a low-pass filter, and is frequently used to reduce aliasing in Fourier transforms [21]. Low-pass filtering is a process used for smoothing or blurring an image. A commonly known low-pass filter is the Gaussian blur in graphics manipulation packages, the effects of which can be seen in Figure 3.7:

The result of using Equation 3.4 (the authors call it the ripple function) as a periodic phase shift filter is that the input noise texture is blurred in the direction of the vector field, simulating apparent motion [2].

Since the LIC algorithm is a local operation, Equation 3.4 must be limited to local extent. The side effect of the localization, however, is observed in the form of abrupt cutoffs in animations that vary the phase $\beta$ as a function of time [2] (top row of Figure 3.8).

The solution is to multiply Equation 3.4 by a Gaussian window function, so the ends of the ripple curve have zero heights. The authors point out that the Hanning

Figure 3.7 A computer-generated graphic of a tree before (left) and after applying Gaussian blur. The filtered version appears to have less detail, especially around the edges of the tree. Image courtesy of Jared D. Rapp, University of Montana.

function itself has windowing properties similar to a Gaussian window function. They define the Hanning windowing function as follows:

$$\frac{1}{2}\left[1 + cos(c\omega)\right] \tag{3.5}$$

where $c$ is a dilation constant.

The kernel function $k$ is the product of Equations 3.5 and 3.4, and can be expressed as:

$$\begin{aligned} k(\omega) &= \frac{1 + cos(c\omega)}{2} \times \frac{1 + cos(d\omega + \beta)}{2} \\ &= \frac{1}{4}\left[1 + cos(c\omega) + cos(d\omega + \beta) + cos(c\omega)cos(d\omega + \beta)\right] \end{aligned} \tag{3.6}$$

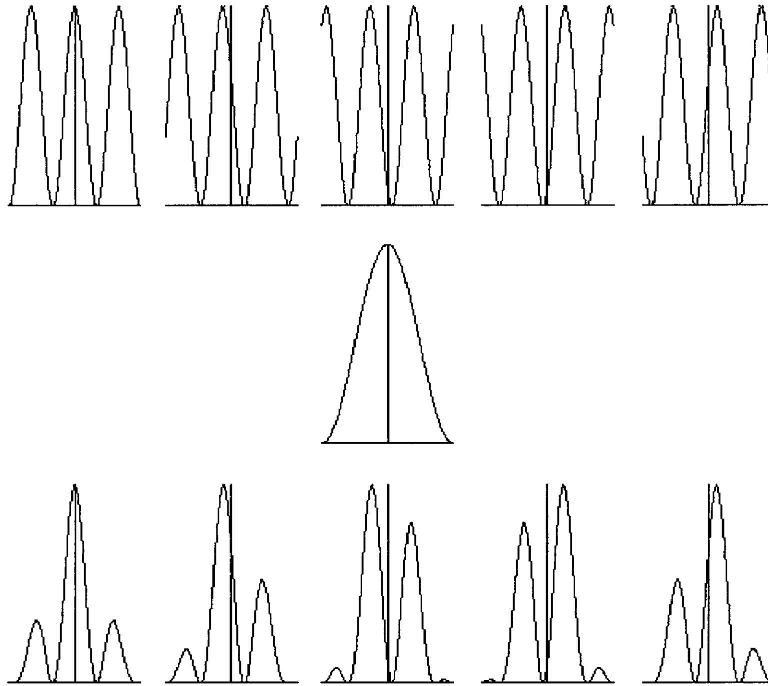


Figure 3.8 Phase-shifted Hanning ripple functions(top), a Hanning windowing function(middle), and their product (bottom). Figure from [2].

The window function (Equation 3.5) has a fixed period of $2\pi$ when $c = 1$ [2].

As shown by the authors in [2], the exact integration of the kernel function $k$ is computed as follows:

$$\int_a^b k(\omega)d\omega = \frac{1}{4}\left(\begin{array}{l} b - a + \frac{sin(bc)-sin(ac)}{c} \\ + \frac{sin(bd+\beta)-sin(ad+\beta)}{d} \\ + \frac{sin(b(c-d)-\beta)-sin(a(c-d)-\beta)}{2(c-d)} \\ + \frac{sin(b(c+d)+\beta)-sin(a(c+d)+\beta)}{2(c+d)} \end{array}\right) \tag{3.7}$$

### Computing Output Pixel Value

Using the exact integration result for the weight $h_i$, the convolution result is computed by using a summation technique:

$$F_{out}(x, y) = \frac{\sum_{i=0}^{l} F_{in}(P_i)h_i + \sum_{i=0}^{l'} F_{in}(P'_i)h'_i}{\sum_{i=0}^{l} h_i + \sum_{i=0}^{l'} h'_i} \tag{3.8}$$

where $F_{out}(x, y)$ is the output value at pixel $(x, y)$, $F_{in}(P_i)$ is the color value from the noise texture at position $P_i$ in the positive, or $P'_i$ in the negative direction in the streamline. $l$ and $l'$ are the distances in the positive and negative direction respectively, and $h_i$ and $h'_i$ are the weighting variables [13].

In essence, the output value of each pixel is computed by the summation of the product of each pixel $P_i$ in the streamline by its weight $h_i$. The denominator in Equation 3.8 is used to normalize the output pixel value [2].

## Euler Method

The Euler method is a simple and popular method for numerical integration of first order differential equations. It is based on the notion that the velocity $v(t)$ at time $t$ is the result of the derivative of the position $y(t)$ at time t:

$$v(t) = \frac{dy(t)}{dt} \tag{3.9}$$

where $v(t)$ is the velocity, and $y$ is the current position at time $t$. The next position at time $t + \Delta t$ is estimated as follows:

$$y(t + \Delta t) = y(t) + v(t)\Delta t \tag{3.10}$$

The Euler method shows satisfactorily results as long as the time step $\Delta t$ is small enough. As $\Delta t$ gets larger, the accuracy of the end result is lessened. The decreasing accuracy of this method is explained by the following:

1. The Euler method computes the rate of change (or the slope) of y and assumes that it is the same throughout the time interval $t$.

2. If, however, the slope changes during the time interval, the change is not taken into account during computation, and discrepancy occurs between the numerical estimate and the exact solution [5][4].

The strategy for minimizing the discrepancy is to choose a sufficiently small step size. The accumulated error in one time step is of order $(\Delta t)^2$, making the Euler method an example of first-order method. Furthermore, the Euler method is asymmetrical, since it uses derivative information only at the beginning of the time interval [5][5].

## Runge-Kutta Method

A more accurate method for numerical evaluation of differential equations is the Runge-Kutta method. This method is based on the Euler method in the sense that it uses the derivative at the beginning of the interval. That derivative, however is used to estimate the slope value at the midpoint of the interval. Finally, the estimated midpoint value is then used to compute the new position $y_{n+1}$, resulting in a better estimate:

[4]Section 2.2, pp. 13–14

[5]Appendix 5A, pp. 120–125

$$k_1 = f(y_n, t_n)\Delta t \tag{3.11}$$

$$k_2 = f(y_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t \tag{3.12}$$

$$y_{n+1} = y_n + k_2 + O(\Delta t^3) \tag{3.13}$$

The error term indicates that this method is of second order (a method is of $n$th order if its error term is $O(\Delta t^{n+1})$ [9]).

An even more accurate variation of the Runge-Kutta method is the *fourth-order* Runge-Kutta algorithm, where the derivative is computed at the beginning of the interval, twice at the middle of the interval, and again at the end of the interval [5]:

$$k_1 = f(y_n, t_n)\Delta t \tag{3.14}$$

$$k_2 = f(y_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t \tag{3.15}$$

$$k_3 = f(y_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2})\Delta t \tag{3.16}$$

$$k_4 = f(y_n + k_2, t_n + \frac{\Delta t}{2})\Delta t \tag{3.17}$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.18}$$

From Equation 3.18 is evident that the slope values estimated at the middle of the interval are given twice the weight than the numerical estimates at the end of the interval.

## Numerical Methods Evaluation

Each of the numerical methods has advantages, as well as shortcomings. That is more true for the Euler method—the method is simple to understand and easy to implement, but the accumulated error increases if the time step $\Delta t$ is not sufficiently small. Moreover, it has been noted that this method is not stable enough in the case of using LIC on circular flow data (the negative effect being that the Euler method produces a spiral flow, rather than a circular pattern).

The Runge-Kutta method is proven to provide better numerical estimation, but it comes at a higher computational cost—the second and fourth order variants perform 2 and 4 function evaluations respectively.

Figure 3.9 illustrates the fall of an object (particle), where air resistance is neglected for simplicity. The Runge-Kutta variants are visibly more accurate than the Euler method:

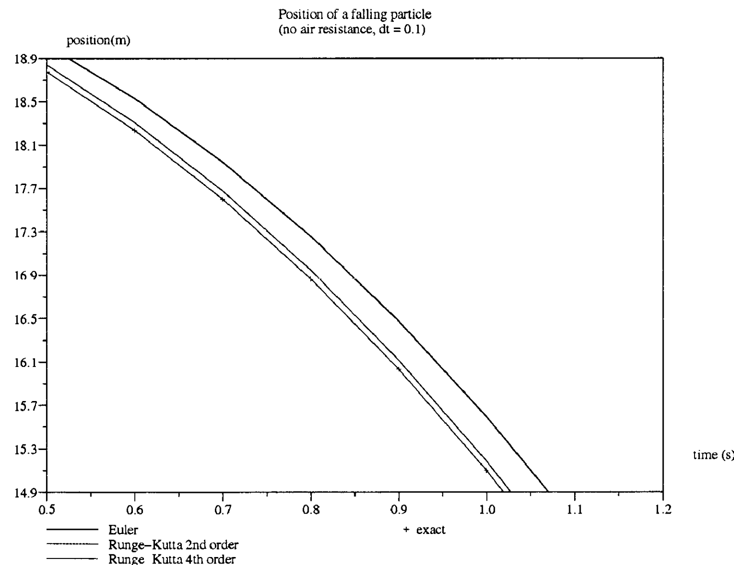



Figure 3.9 Accuracy for the numerical methods. The two Runge-Kutta variants are closer than the Euler method to the exact solution.

Summarized below are the accumulated error and number of function evaluations for each numerical method. It is evident that the Runge-Kutta variants are slower than the Euler method, since they require a higher number of function evaluations. Those additional function evaluations, however, result in better estimates (Figure 3.9).

| Method | Error | Function evaluations |
|---|---|---|
| Euler | $O((\Delta t)^2)$ | 1 |
| RK 2 | $O((\Delta t)^3)$ | 2 |
| RK 4 | $O((\Delta t)^5)$ | 4 |

Table 3.1 Accumulated error and number of function evaluations comparison for the numerical methods.

## Alpha Blending

Alpha blending is a process of creating the effect of transparency, and is frequently used in computer graphics [1]. The technique can be described as overlaying a translucent color layer on top of a background image, creating a blending effect:

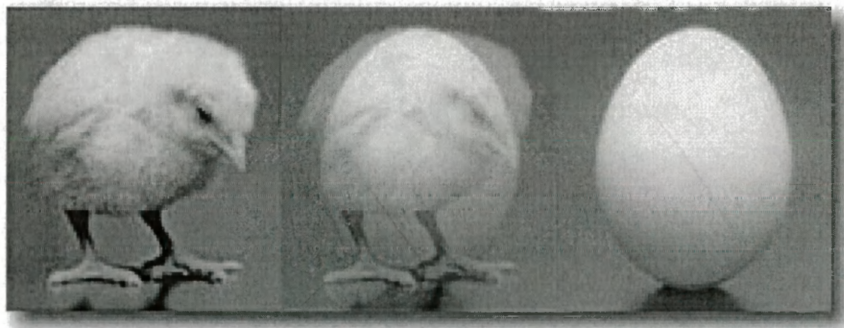


Figure 3.10 An example of alpha blending with alpha values of 0, 0.5, and 1 respectively. Image courtesy of Duane Bong, visionengineer.com.

Its significance for data visualization is that it allows for the display of a multitude of data components within the same image window.

In most computer graphics interfaces, there are 4 channels used to define color. The first 3 channels are used to describe the red, green and blue colors, while the fourth channel (the alpha channel) describes the level of transparency. Furthermore, this channel specifies how the foreground colors should be merged with those in the background when overlaying occurs [1].

As Bong describes in [1], the method for calculating alpha blending is:

$$[r, g, b]_{blended} = \alpha[r, g, b]_{forergound} + (1 - \alpha)[r, g, b]_{background} \tag{3.19}$$

# CHAPTER 4 IMPLEMENTATION AND RESULTS

## Implementation Plan

### Initial Prototype

In the beginning phase of my research, I experimented with prototyping work, and studied existing code to gain understanding for the LIC algorithm (*Integrate and Draw* in [11], and a C++ implementation in [7]) The result of my early prototype was simply an algorithm that averaged the color of all pixel values in a single line. Later, the same averaging technique was restricted to a desired streamline length. Finally, I experimented with computing a streamline (Formula 3.1), and the workings of the kernel function (Equation 3.6). The result was a functioning prototype, written in the Perl language. Perl was selected because it is a scripting language, and it runs on different platforms. A LIC image produced by the Perl prototype is seen in Figure 3.1.

### Interface Design

Initial requirements for the OpenDX LIC module were derived from the general description of the LIC algorithm, as described by Cabral and Leedom in [2]. Therefore, the module had to accept a noise field and a vector field, and had to produce a field object, showing the flow lines of the vector field.

Consequcntly, it was determincd that thc uscr should also specify desired length,

as well as the method[1] for computing the streamline. In addition, user-defined values for the inputs kernel function would allow for greater control over the output image, as well as the sense of animation. Another important aspect is to allow the user to control the alpha channel for the color, thus defining the opacity level. As a result of these observations, the user interface for the LIC module was defined as follows (see also Figure 4.1):

| **Tab name** | **Type** | **Data type** | **Description** |
|---|---|---|---|
| noise | input | field | the white noise texture |
| xy vector | input | vector list | the vector field containing x and y components |
| length | input | integer | desired streamline length (default is 10) |
| method | input | string | desired method for computing streamlines (Euler or RK2) |
| opacity | input | scalar | desired opacity level for the output image (default is 0.75) |
| phase shift | input | scalar | current kernel phase shift (default is 3.14) |
| texturized flow | output | field | the computed LIC image |
| opacity | output | scalar | specified opacity level |

Table 4.1 Interface design for the OpenDX LIC module.

## Implementation

As most other OpenDX modules, the LIC module is written in the C programming language. The DX API was also used for error and type checking, memory management, using existing data types and operations (*e.g.,* Points and Vectors), as well as defining new types suitable for the LIC algorithm (*e.g.,* Pixel). The current implementation is in the form of a runtime-loadable module (Table 2.1) for the Linux

[1]See Euler and Runge-Kutta methods in Chapter 2.

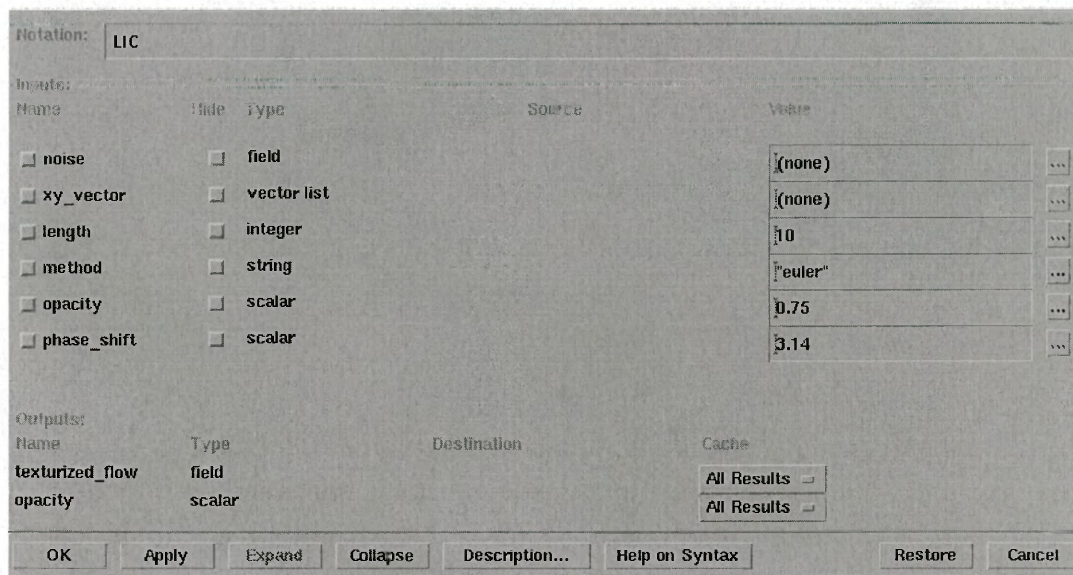



Figure 4.1 Inputs to the LIC module can be specified, as well as routed from other modules.

operating system. The module is not a part of the default OpenDX module list, but it can be loaded from the command line, as well as the VPE window.

The module inputs are the same as defined in the previous section, as well as the outputs (Table 4.1 and Figure 4.1). The LIC module has a similar wired tab interface as all other DX modules, allowing the user to focus on visualization and reuse of the module between DX applications (Figure 2.8).

## Test Suite

In order to test the LIC module, various vector field matrices were used. To ensure validity of the results and reliability of the LIC module, it was decided that matrices should vary in their proportions (square vs. rectangular), majority order (row vs. column), and sizes. As a result of these criteria, the list of test datasets was defined

as follows:

| Vector Data | Grid size | Majority | Description |
|---|---|---|---|
| Peaks | 200x200 | Column | A vector field of equal size in the $x$ and $y$ directions, generated by the Peaks function in Matlab. |
| IceView | 101x101 | Row | The velocity of of an ice field. |
| Gwenn Flowers | 150x107 | Column | Glacier flood analysis data, as a result of volcanic activities in Iceland. |

Table 4.2 Test suite for the OpenDX LIC module.

## Results

In its current implementation, the LIC module can produce a texturized flow of an arbitrary vector field. Streamlines are computed for a user-specified length, where the optimal value is usually $10 - 15\%$ of the width of the vector field. The LIC module requires as inputs a noise field and a vector field of the same size in each dimension. Other inputs, such as the streamline length, opacity level and a numerical integration method for computing streamlines are optional. Comparison of the LIC module to traditional vector visualization techniques can be seen in Figures 4.2 and 4.3:

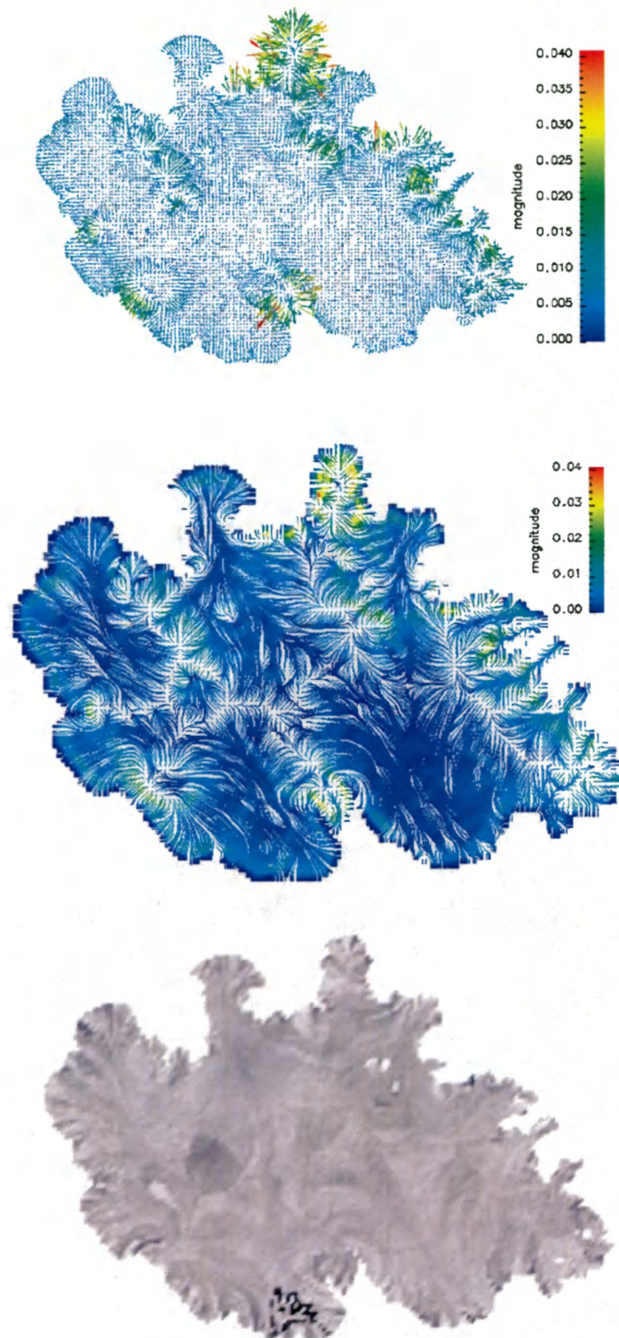



Figure 4.2 Gwenn Flowers' glacial flood data. Shown are glyphs (top), where arrows are scaled to velocity magnitude, streamlines (middle), and the texturized LIC flow (bottom).

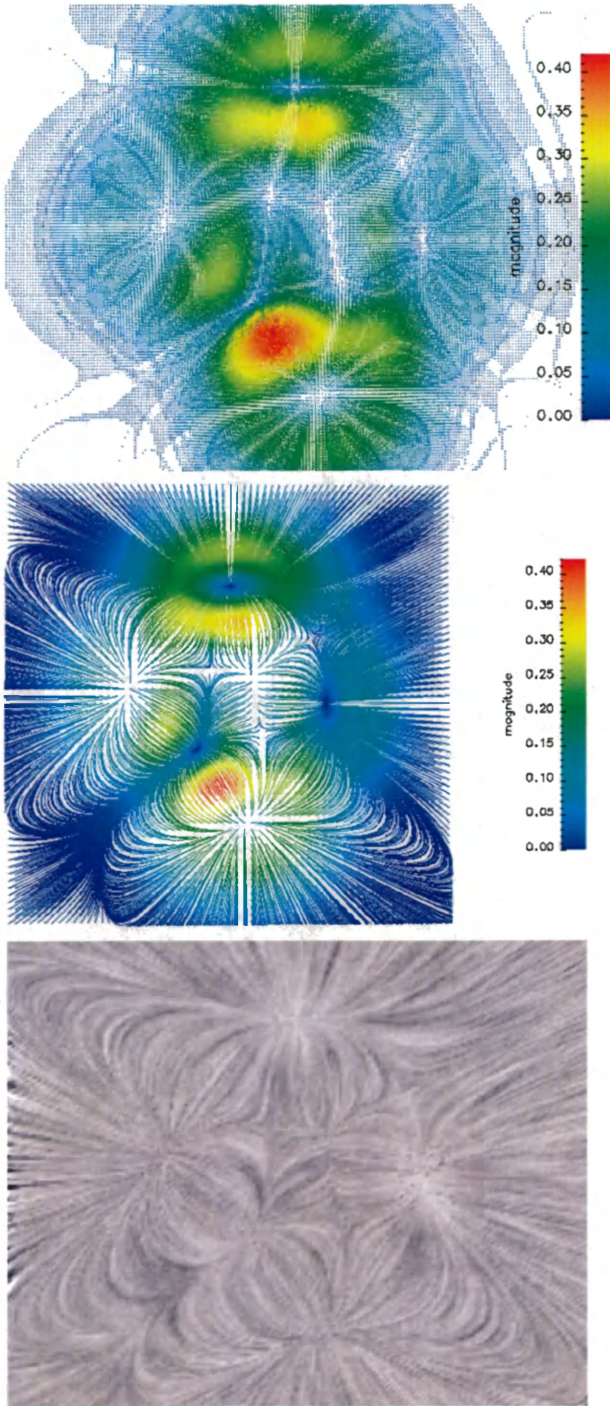



Figure 4.3 Matlab Peaks: comparing glyphs (top), streamlines (middle), and LIC (bottom).

### User Control

The configuration panel of the LIC module (Figure 4.1) allows for further control and finer adjustments to the resulting image. For example, altering the length of the computed streamlines, as well as manipulating the phase shift $\beta$ (Equation 3.6) can lead to better results:

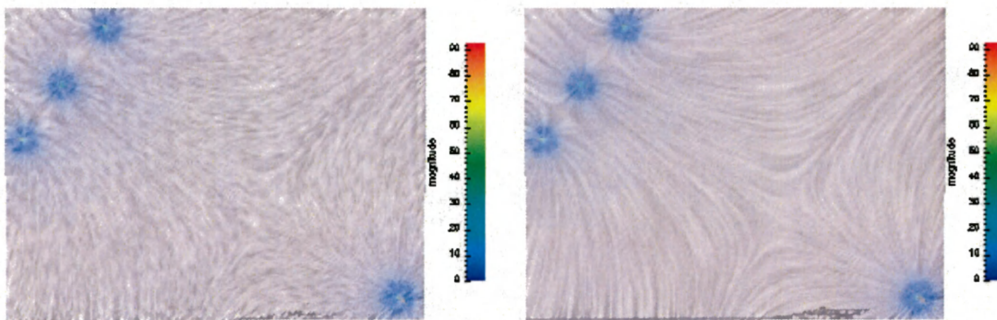


Figure 4.4 An electric field, visualized with the LIC module. The two images differ as a result of altering the streamline length and kernel function parameters.

### Alpha Blending

Additional benefits of the LIC implementation for OpenDX include the ability to overlay multiple data components. More specifically, the texturized vector flow can accept a user-specified opacity level, so underlying elements remain visible (Figure 4.5).

### Animation

The capability of OpenDX to visualize time series data can be used to build animations. Even when time series data is not present, altering the values to the kernel

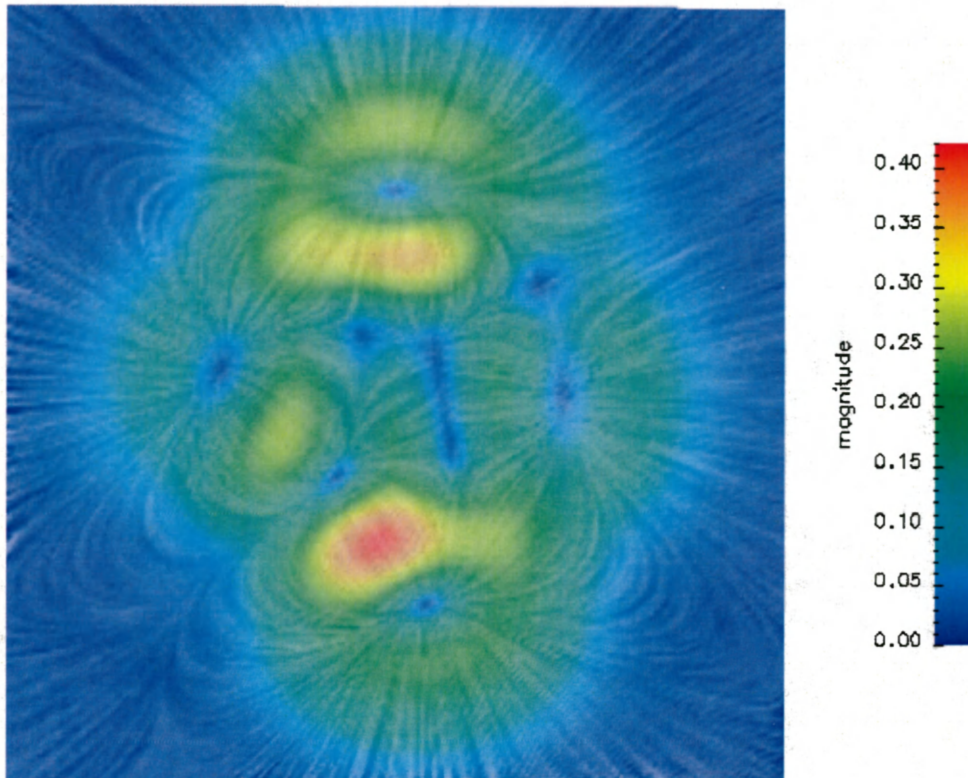



Figure 4.5 The result from the LIC module for OpenDX , superimposed over the magnitude of the vector field. The added opacity level enables the viewer to see the color-coded magnitude, as well as the flow of the vector field.

function of the LIC module can create visual sense of movement. For example, when coupled with a *sequencer* or a *foreach* module, LIC will display a series of texturized images, in which lines seem to be flowing. As a result, the problem of directional loss in static LIC images is remedied.

### Invalid Positions

The problem of white noise traces in the texturized flow is addressed by marking positions with zero $x$ and $y$ vector values as invalid. The solution produces a better

overall image, one that shows only the visualized flow, without remnants of input noise texture spots (Figure 4.6).

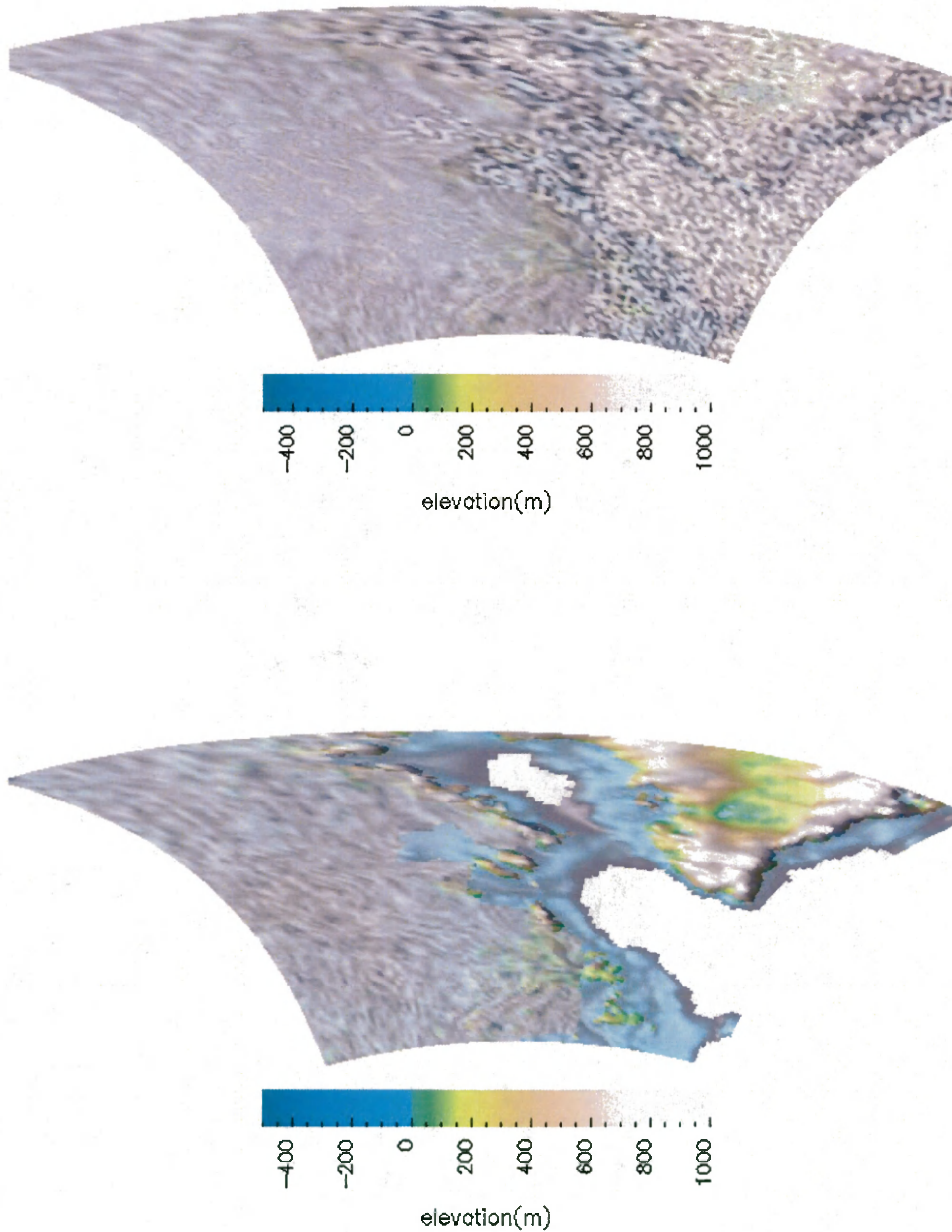



Figure 4.6 Texturized velocity flow of an ice field, superimposed over the topography of North America and Greenland, before and after invalidating positions with zero vector values.

# CHAPTER 5 CONCLUSIONS AND FUTURE DIRECTIONS

## Conclusions

With their LIC algorithm, Cabral and Leedom [2] have influenced many researchers. For example, Wegenkittl, and Gröller authored oriented LIC (OLIC) and fast oriented LIC (FROLIC), which are variants with a directional component [20], Stalling and Hege developed a fast and resolution independent LIC [14], and a pseudo-LIC algorithm (PLIC) by Verma, Kao, and Pang [18].

The work in this thesis has been similarly inspired by Cabral and Leedom. The result, however, is new and unique, given the ability to generate transparent LIC images and animate them. As an OpenDX module, the LIC implementation described in this thesis has numerous advantages over standalone implementations, or those included in graphics packages. The results of the OpenDX implementation include:

1. **Integration with a large visualization suite:** The LIC module adds an alternative for visualizing vector fields in OpenDX . The integration in itself reveals additional benefits, some of which are:

   (a) **A well-defined user interface:** Similar to all other OpenDX modules, the LIC module has input/output tabs, which can be wired from or into other modules (Figure 2.8). The LIC configuration panel allows the user to

have finer control over the module parameters, by changing default values (Figure 4.1).

(b) **Interaction with other modules:** The output from the LIC module can be routed to other modules for further manipulation. For example, modules such as Regrid can be used to change the number of data points.

(c) **Rapid expansion:** LIC can be combined with any other visualization method, for example:

i. **Rubbersheet:** Although the current LIC implementation is restricted to two dimensions, the Rubbersheet module can create a 3-dimensional view of the data (Figure 4.5).

ii. **Colors and opacity:** Various color modules (*e.g.,* Color, or Auto-GrayScale) can be used to add color to the LIC output.

iii. **Multiple superimposed data layers:** Given OpenDX's capability to display multiple data elements and add opacity, the LIC output can be superimposed on additional data layers (or vice versa).

iv. **Traditionally displayed vector data:** The LIC module can be used in conjuction with existing methods of displaying vector data. For example, a texturized flow can be combined with glyphs or streamlines.

v. **Ability to animate:** Given OpenDX's facilities to represent time series data as sequence of images, the LIC module can be used to produce animations. In addition, altering parameters of the kernel function $k$ (Equation 3.6) can produce a realistic sense of movement even when time series is not present.

2. **Open source module:** The source code for the LIC module is freely available. Unrestricted source code access has been given for a number of reasons. This

thesis has been based largely upon existing work. Thus, source code availability conforms to long-standing traditions of unobstructed exchange of knowledge and ideas amongst researchers. Another reason is that it allows others to offer their collective insight by the means of improving or extending the source code. The intended results are that the LIC module will be continuously updated in a constantly-developing scientific environment, and that any derived work will be made freely available. Lastly, binary versions can be compiled for other platforms.

The LIC algorithm has proven suitable for dense vector fields. This method eliminates the necessity to subsample data in order to reduce clutter (Figure 2.6). The LIC module is a viable alternative to existing methods in OpenDX for visualizing vector data—one that can be used to generate continuous flow and eliminate the loss of data traits as a result of subsampling.

The LIC module can be used in IceView to produce a texture-based flow of the velocity of ice masses. In addition, the ability to add opacity level to the velocity layer will ensure that other layers, such as the continental bed or basal temperature conditions, remain visible.

Other possible applications include the modeling of weather systems, computational fluid dynamics, and electro-magnetic fields.

## Known Limitations

Currently the LIC module for OpenDX compiles only on the Linux platform. Due to the limited availability of the most recent installed OpenDX version in the CS lab at The University of Montana, the LIC module was developed on the Linux platform.

Besides its ability to show the flow for a vector field in great detail, the result might be misleading to the user. More specifically, the one-tone color (*e.g.,* gray scale) could be misinterpreted to mean indication of field strength (magnitude). In fact, it simply distinguishes between computed streamlines.

As in the original algorithm, this implementation is limited to two dimensions. However, OpenDX does facilitate the creation of 3-dimensional images from 2-dimensional data (*e.g.,* using the Rubbersheet module).

As described in the analysis of the LIC algorithm in Chapter 3, the module depends on an input noise texture (see also Figure 4.1). In the case of visualizing time series data, OpenDX program execution follows for each time frame. To prevent generating a new noise texture with each execution, a noise texture for one time step is saved first and later imported. The workaround insures that the same random texture is used for executing the LIC module during each consecutive time step.

## Future Directions

Perhaps the most logical improvement to the current version of the LIC module is to add a color component to the generated flow. This could be in the form of generating a discrete color for each streamline in the vector field, as described and implemented by the author of *Integrate and Draw* [11]. Moreover, the *Integrate and Draw* implementation eliminates random noise texture dependence. In the case of time series data, a similar approach will simplify the visualization process and improve usability.

The LIC module can be extended to produce 3-dimensional flow. Such improvement will allow for true visual representation for some practical applications (*e.g.,* weather storms).

The objective evaluation of the effectiveness of this texture-based approach with respect to icon-based and streamline techniques is currently absent. The conducting of user evaluation studies is one way to quantify the usefulness of the convolution method.

The LIC module does not have to be restricted to the brute-force approach for computing streamlines, as in the original method. Experimenting with other variants of the LIC algorithm, such as fast LIC [14] and FROLIC [20] will lead to speed optimizations.

# BIBLIOGRAPHY

[1] Duane Bong. *Alpha blending.* Vision Engineer, http://www.visionengineer.com/comp/alpha_blending.shtml, 1999–2004.

[2] B. Cabral. Imaging vector fields using line integral convolution. In *Computer Graphics Proc.*, pages 263–270, 1993.

[3] IBM Corporation. *IBM's Visualization Data Explorer Programmer's Reference.* IBM Press, http://opendx.npaci.edu/docs/html/pages/progu344.htm, 1996-2005.

[4] IBM Corporation. *IBM's Visualization Data Explorer User's Guide.* IBM Press, http://www.research.ibm.com/dx/docs/legacyhtml/usrguide.htm, seventh edition, May 1997.

[5] Harvey Gould and Jan Tobochnik. *An Introduction to Computer Simulation Methods.* Addison-Wesley, Reading, Massachusetts, second edition.

[6] David N. Kenwright and Gordon D. Mallinson. A 3-d streamline tracking algorithm using dual stream functions. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 62–68. IEEE Computer Society Press, 1992.

[7] Harper Langston. *Fluid Dynamics Visualization.* NYU Media Research Lab, http://mrl.nyu.edu/~harper/stokes/index.html, 2002–2005.

[8] Lynh H. Loomis. *Calculus.* Adison-Wesley, Reading, Massachusetts, third edition, 1982.

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, section 16.1, pages 710–714. Cambridge University Press, 1992.

[10] Laurie Hodges Reuter, Paul Tukey, Laurence T. Maloney, John R. Pani, and Stuart Smith. Human perception and visualization. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 401–406. IEEE Computer Society Press, 1990.

[11] Carlos Pérez Risquet. Visualizing 2d flows: Integrate and draw. In *Proceedings of the 9th EUROGRAPHICS Workshop on Visualization in Scientific Computing*, 1998.

[12] Matthew W. Rohrer. Seeing is believing: the importance of visualization in manufacturing simulation. In *Proceedings of the 32nd conference on Winter simulation*, pages 1211–1216. Society for Computer Simulation International, 2000.

[13] H. Shen and S. Bryson. Using line-integral convolution to visualize dense vector fields. *Computers in Physics*, 11(5):474–478, Sep/Oct 1997.

[14] Detlev Stalling and Hans-Christian Hege. Fast and resolution independent line integral convolution. *Computer Graphics*, 29(Annual Conference Series):249–256, 1995.

[15] D. Thompson, J. Brown, and R. Ford. *OpenDX Paths to Visualization.* VIS, Inc, Missoula, Montana, 2000.

[16] Edward R. Tufte. *The visual display of quantitative information.* Graphics Press, Chesire, CT, 1986.

[17] Jarke J. van Wijk. Spot noise texture synthesis for data visualization. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 309–318. ACM Press, 1991.

[18] Vivek Verma, David Kao, and Alex Pang. PLIC: Bridging the gap between streamlines and LIC. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 341–348, San Francisco, 1999.

[19] Colin Ware. *Information visualization: perception for design.* Morgan Kaufmann Publishers Inc., 2000.

[20] R. Wegenkitt and E. Groller. Fast oriented line integral convolution for vector field visualization via the internet. In *Proceedings of the 8th conference on Visualization*, pages 309–316, 1997.

[21] Eric W. Weisstein. *Hanning Function.* MathWorld–A Wolfram Web Resource, `http://mathworld.wolfram.com/HanningFunction.html`, 1999–2005.