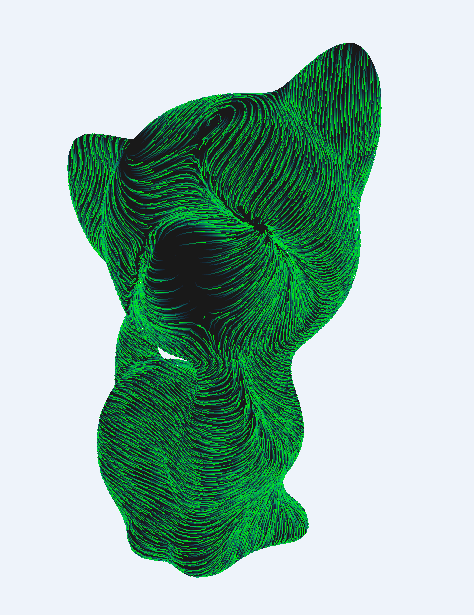
Vector Field Viewer Project Report



### Authors:

|  |  |
| --- | --- |
| Dmitry Ogurtsov | ogurdima@gmail.com |
| Yacov Glushkin | yacov.gls@gmail.com |

### Instructor:

|  |  |
| --- | --- |
| Miri Ben-Chen | mirela@cs.technion.ac.il |

Contents

[Project Description 3](#_Toc421535779)

[Project Goal 3](#_Toc421535780)

[Inputs 3](#_Toc421535781)

[Outputs 3](#_Toc421535782)

[Additional Details (based on specific implementation) 3](#_Toc421535783)

[Prerequisites 3](#_Toc421535784)

[Features 3](#_Toc421535785)

[Field file syntax 4](#_Toc421535786)

[Constant Field 4](#_Toc421535787)

[Variable Field 4](#_Toc421535788)

[Implementation Environment 5](#_Toc421535789)

[Overview 5](#_Toc421535790)

[Solution Structure 5](#_Toc421535791)

[Project Directory Structure 5](#_Toc421535792)

[Implementation Details 6](#_Toc421535793)

[High Level Partial Class Diagram of VectorFieldViewer Project 6](#_Toc421535794)

[High Level Program Flow 7](#_Toc421535795)

[Communication Flow 7](#_Toc421535796)

[C++ and .NET Interoperability 8](#_Toc421535797)

[Particle Path Computation Algorithm Details 8](#_Toc421535798)

[The Basic Algorithm 8](#_Toc421535799)

[Problem 0: Field representation 8](#_Toc421535800)

[Problem 1: Field computation 9](#_Toc421535801)

[Problem 2: Field points out of face 10](#_Toc421535802)

[Problem 3: Intersection testing 10](#_Toc421535803)

[Problem 4: Dense sub paths 10](#_Toc421535804)

[Problem 5: Particle path computation is slow 11](#_Toc421535805)

[Particle Paths Drawing Details 11](#_Toc421535806)

[Basic algorithm for finding the visible part of the path and interpolating colors 12](#_Toc421535807)

[The visualization loop 12](#_Toc421535808)

[Particle movement continuity 12](#_Toc421535809)

[Residual Problems and Limitations 12](#_Toc421535810)

[Summary 13](#_Toc421535811)

# Project Description

## Project Goal

Provide a utility for visualization of time-dependent vector fields on meshes in a three dimensional space. A field is defined on surface of mesh only. The field will be visualized by tracing paths of particles spawned on the mesh initially. Particle will take into account field strength and direction at given point and move accordingly. The particle will leave a trail. For fields that are constant in time particle’s trail will coincide with a field line of the aforementioned field. For time-dependent fields trail of particle has no mathematical meaning.

## Inputs

* A mesh file in object file format (.off)
* A field file in field-per-face format (to be specified in implementation)
* Visualization-specific configuration parameters (to be specified in implementation)

## Outputs

* Interactive (rotate,translate,zoom) window which allows user to see the particles move according to vector field
* Screenshots in accepted image file format on disk (on demand)

# Additional Details (based on specific implementation)

## Prerequisites

* Windows platform
* OpenGL v2.0
* .NET v4.0

## Features

* 32 and 64 bits support, latter required for detailed visualization of large models
* Mesh loading in Object File Format
* Support for constant field or time-variable field expressed in terms of constant field snapshots
* Smooth field transition across different mesh faces achieved by translating internally field-per-face into field-per-vertex representation and interpolating vertex values
* OpenMP based parallel simulation
* Fixed Stack OpenGL real time visualization
* Powerful GUI control panel that allows controlling simulation and visualization parameters
* Support of visualization snapshots in PNG format

## Field file syntax

### Constant Field

* File extension must be .vf
* All lines are the same
* Each line represents a vector in 3D space
* Number of lines must be equal to number of faces of (already loaded) mesh
* Example:  
  0.008760250 0.004253466 0.005110653  
  0.021301756 -0.001675812 -0.004383483  
  ……

### Variable Field

* File extension must be .txt
* All lines are the same
* Each line represent a snapshot of vector field in a specific time
* Format: <time> <filePat h>
* filePath is path to file with Constant Field syntax
* Example:  
  0.001 frog\_s5\_0.001.vf  
  0.03 frog\_s5\_0.026.vf

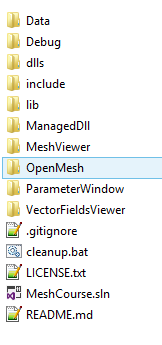
# Implementation Environment

## Overview

The project implemented for windows OS, using Microsoft Visual Studio 2012. MeshCourse.sln is the solution file for the project.

## Solution Structure

The solution contains 5 projects:

* OpenMesh
  + Provides DCEL data structure for storing and manipulating meshes
  + Open source project
  + Taken as-is, no modifications needed
* MeshViewer
  + Provides basic openGL viewer that works with OpenMesh
  + Contains a base class for VectorFieldViewer
* ParameterWindow
  + Provides configuration widget for VectorFieldViewer
  + Written in .NET (C# + WPF)
* ManagedDll
  + Provides a communication bridge between VectorFieldViewer and ParameterWindow
  + Written in managed (.NET) C++
* VectorFieldViewer
  + The main project.
  + Provides
    - Particle paths construction algorithm
    - Particle paths visualization

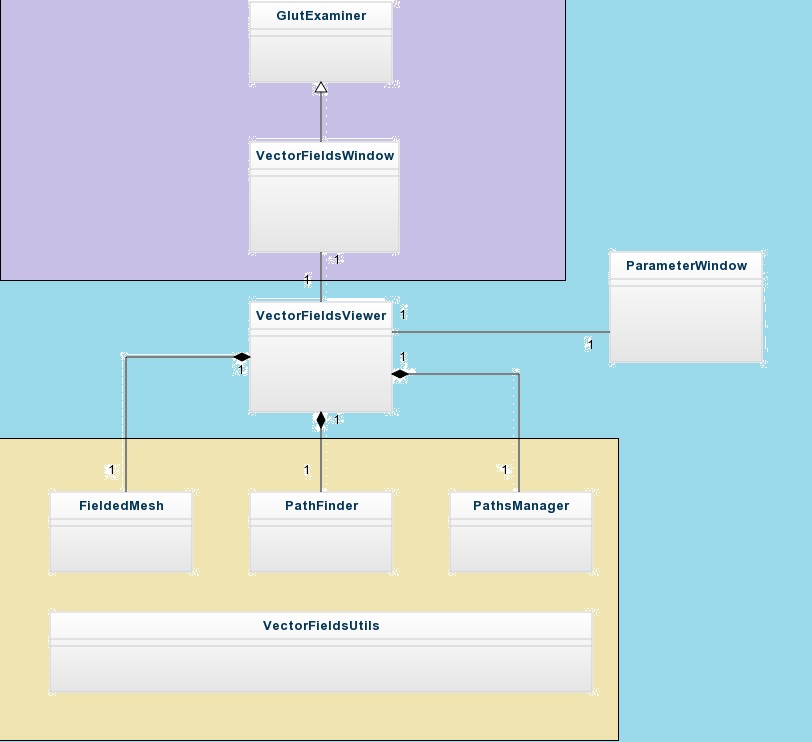
## Project Directory Structure

Each project described above is in separate folder with matching name. In addition to that, project’s root contains the following folders:

* Data
  + Contains samples if Mesh and Field files
* dlls
  + Contains dll files that are consumed by the project
  + Relevant dll files are copied to binary folder upon every build
* include
  + Additional include files go here
* lib
  + .lib files of third party libraries go here

# Implementation Details

## High Level Partial Class Diagram of VectorFieldViewer Project



The diagram displays relations between major players in the project. First we need to mention that GlutExaminer and ParameterWindow are not in VectorFieldViewer project. But for the sake of clarity of explanation we will discuss this diagram as if all players are in the same project.

Purple part is responsible for OpenGL drawing and Windows related activities (allocating window, creating context, etc). VectorFieldsWindow class is used to decouple VectorFieldsViewer from OpenGL draw details. It is the main class that deals with **vector field drawing**. It creates VectorFieldsViewer, registers for its events and changes view according to them.

Blue part contains high level classes used for **vector field computation** specifically:

* VectorFieldViewer class loads and stores mesh and vector fields, spawns particles, and computes their paths.
  + Uses ParameterWindow by invoking functions that are provided by an external .dll
  + Raises events (calls functions that are registered via function pointers)
* ParameterWindow is a .NET GUI control panel that allows user to configure VectorFieldsViewer

Yellow part is (almost) purely algorithmic. As was said before, VectorFieldViewer is the central high-level class used for computing vector field paths. For this purpose it uses three major helper classes:

* FieldedMesh
  + In memory representation of a fielded mesh
  + Uses OpenMesh functionality to store geometric data
  + Associates finite number of field snapshots to each vertex in mesh
* PathFinder
  + Based on initial particle position and FieldedMesh computes particle’s path
* PathsManager
  + Stores computed particle paths in a data structure that allows efficient copy of graphical data to GPU memory by OpenGL

In addition, all mentioned above use utility class VectorFieldsUtils, which exposes static helper functions

## High Level Program Flow

1. Create and initialize an instance of VectorFieldsWindow
2. Initialize OpenGL context, load necessary dlls, open interactive window
3. Create an instance of VectorFieldsViewer, register to its events
   1. Initialize the scene
4. Open parameter window, bind configurable parameters with an instance of VectorFieldsViewer
5. Wait for input in glut main loop

## Communication Flow

Despite the complicated program structure, the communication between major players in the project is described quite simply

VectorFieldsWindow

VectorFieldsViewer

ParameterWindow

Function call

Function call

Callback

Callback

## C++ and .NET Interoperability

ParameterWindow is a WPF control written in C#. VectorFieldsViewer (and almost everything else) is written in native C++, native meaning C++ with pointers and actual memory addresses. This does not fit well with .NET’s managed environment, with high abstraction, security and no memory addresses. Interoperability between the two can be tricky.

Our requirement was that we will be able to call C# methods from native C++ (e.g. creating configuration window, updating “current time” entry), but also register C++ methods as callbacks from C# code (e.g. when user presses button in control window our C++ code must react accordingly). There is a couple of ways to achieve this two-way interoperability. We have chosen (for simplicity mainly) to use managed C++ bridge.

ManagedDll project in the solution is this bridge. It is a C++ dll project that exposes several methods. These methods can be called from native C++ (VectorFieldsViewer) directly. Internally, ManagedDll marshals received arguments to C# dll (ParameterWindow) which it uses. This is possible because ManagedDll is written in managed C++, so .NET knows how to transfer values to and from C#.

As a matter of fact, because native C++ consumes ManagedDll as dll, our control panel implementation can be replaced with anything else, as long as the new implementation satisfies the communication contract described in file External.h

## Particle Path Computation Algorithm Details

In order to display paths of particles we need to compute these paths first. This seems like a simple task, but actually we encountered a lot of problems. In the next section we will present an overview of the algorithm, together with problems that arise and the way we solved them.

### The Basic Algorithm

1. At each step let *t* represent current time, *loc* represents current particle location
   * Initially a particle is spawned in the middle of each face
2. Compute vertex field in *loc* at *t*.
3. Move the particle to the next location according to the vector field. Vector field is treated as speed of the particle at that time, the time difference is *dt*, so the next *loc := loc + v\*dt*
4. Check whether we crossed the boundary of a face.
   * If so move the particle back to the intersection point, compute *tintersection. t := tintersection*
   * Otherwise *t := t + dt*
5. If *t* is greater than simulation time upper bound (configurable parameter) finish. Otherwise go to step 1

### Problem 0: Field representation

Even before thinking how to implement the algorithm we need to decide how to store field. The requirement is time-dependent field, field-per-face. So the obvious solution is to store a field in each face as a series of snapshots, each having time and field vector at that time. Then we can get face’s field at any time t by linearly interpolating vectors at closest (in terms of time) snapshots

### Problem 1: Field computation

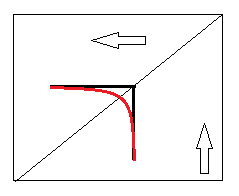
At time *t* at location *loc* we need to compute vector *field(t, loc).* We already know how to compute *field(t)* for each face. How do we compute the total field in *loc*?

#### Solution 1: Weighted Average of all face fields

This approximation is as close to real field as we can get. But it requires *O(n)* operations for each *t* for each particle (and there are *O(n)* of them), which results in *o(n2)* complexity. This solution becomes too slow even for moderate meshes.

#### Solution 2: Weighted Average of l-ring neighborhood

We approximate field value by taking into account only l-ring, assuming other faces are too distant to significantly influence the result. This results in *O(n\*l)* operations for each *t*, which can be viable for small *l*. But there are major problems with this solution:

1. By using l-ring we take connectivity data and treat it as proximity data. In other words, we ignore everything beyond l-ring, even though faces that are there can be very close and have major influence on resulting field
   1. Can be solved by storing proximity data and taking into account not l-ring but closest l faces. As we will see this does not solve the second problem though.
2. When particle location passes from one face to another, the l-ring we are taking into account changes. This change is momentary, so the resulting field changes momentarily. This means that just by tracing the particle path we will get non-continuous line. Red line in the example is what we want to see, black is what we will get
   1. This can be solved by using a smoothing technique with the polyline as a control polygon. But the result can
      1. Differ significantly from the actual field line
      2. Dive inside the mesh or conversely “stick out” too much

#### Solution 3: Weighted Average of neighboring vertices

One can notice that the discontinuity problem disappear once we store field data in vertices instead of faces. Then we can use barycentric coordinates of a point to compute vector field at that point. Because two neighboring faces share an edge (with two vertices), the resulting path will always be continuous (at least G1).

This is the solution we chose. Basically, by changing the solution to Problem 0 we make Problem 1 disappear altogether.

When using this technique, a new question arises: how do we translate field-per-face into field-per-vertex? A number of solutions exist, ranging from taking into account only neighboring faces to taking into account all faces of the mesh. We chose the former for simplicity and efficiency, but any other way will work too. FieldedMesh::assignFieldToVertices() function performs this translation, it can be easily re-implemented to match other solutions.

### Problem 2: Field points out of face

The assumption is that field is defined only on the surface of the mesh, and particle moves only on the surface. But because we take an approximation of the field at point, and not the real field, the value we get can be not in the same plane as the face where the particle is currently in.

We solve this by projecting the weighted average onto relevant face’s plane. Given the approximated field that’s really the best thing we can do.

### Problem 3: Intersection testing

When particle travels along some line in face plane, eventually it reaches an edge of the face. This must be detected by an algorithm, so the question of intersection testing with face edges arises.

There is no “solution” here, we need to do intersection checks every time we detect that next point location is “outside” of the current face. We formulated the following requirements for the intersection test:

* It must be efficient
* It must take into account numerical errors
* It must be deterministic

The first two requirements are (somewhat) satisfied by the code itself, there is no point to explain it here. The third requirement is achieved by the fact that each time we do an intersection test, we go through face’s edges in the same order. This way if a particle hits a vertex, it will always decide that it crossed the same edge (the first in the order of the check)

### Problem 4: Dense sub paths

As it was said before, when particle crosses edge between two neighboring faces (A and B) there should be no continuity problem. It is so, but only for average field value that we get from vertices. But then we project this vector onto plane of the face. And because plane of face A is not the same as plane of face B, the resulting vectors are different this can cause the scenario we see on the illustration. New and new segments are added to the polyline, while *t* is not incremented.

This was solved by adding a convergence test each simulation step. This was implemented as a two-step process:

* Check if previous two points are close enough
  + If so, collapse them into single point, and increase collapse count
  + Otherwise set collapse count to zero
* Check how many consecutive collapses were made. If collapse count is greater than a predefined threshold, we are stuck (the *t* is not incremented enough). Then we increment t manually. Because the field is dependent on time, after the *t* is incremented we may “get out” from the problematic point

### Problem 5: Particle path computation is slow

Again, there is no single “solution” to this problem. We needed to make sure that our implementation of primitive operations during simulation is semi-optimal and that number of such primitive operations is as small as possible.

The point worth mentioning is that paths are independent from one another. So the problem of computing paths is of concurrent nature. Ideally we would want to implement this on GPU, making use of high data parallelism. But, the simulation process has proved to be quite complicated with a lot of extreme cases and branches, and this is undergraduate project after all ☺. So we went with CPU computation + OpenMP instead.

OpenMP allowed us to easily parallelize particle path simulation, reducing the computation time proportionally to the number of CPU cores. The only OpenMP directive is in function PathFinder::getParticlePaths(). OpenMP can be disabled (e.g. for debugging) just by commenting this line out.

## Particle Paths Drawing Details

Now when we have all these paths we need to draw them efficiently. The key point here is to minimize amount of data copied from CPU to GPU in each frame, and, more importantly, minimize number of data transfer calls. It is also important to put the data that will be copied to close memory locations, to utilize processor cache as much as possible. We use PathsManager class for that.

PathsManager stores all paths in a single array, storing together geometry, time and color data. This allows us to copy all the needed data to GPU with only a few OpenGL calls, whilst utilizing CPU cache. VectorFieldsWindow::drawVectorField() function has all the OpenGL code there is to it.

The array is float\* and internally contains each point as 8-vector, storing position, color and time of each point. Using this structure we can provide OpenGL with pointers to vertex and color data for a single polyline:

glVertexPointer(3, GL\_FLOAT, 8 \* sizeof(float), dataArray);

glColorPointer(4, GL\_FLOAT, 8 \* sizeof(float), dataArray + 4);

For polyline with 3 vertices it can be visualized like this:



VertexData

VertexStride

ColorData

ColorStride

NumOfPoints = 3

As you can see, the *time* coordinate of a point is not used by OpenGL. Yet, it is stored in the array. Now it’s a good time to remind how we visualize the field. We are not just drawing field lines, we want to create effect of particle moving on the field line, leaving trail. This is done using color/alpha interpolation across visible part of the particle path. At each point in time *t* we want to display a snapshot of particle locations/trails that is correct for that time, i.e. only a part of the path is visible.

### Basic algorithm for finding the visible part of the path and interpolating colors

1. Assume *t* is current time, *path* is an array of the structure specified above, *trailTime* is particle’s trail “cool down” time – we do not want to see a trail where particle was earlier than *t - trailTime*
2. Find *h* such that *path[h].time <= t* and *path[h+1].time > t*
   1. The path is sorted by time, so there is only one *h* like that
3. Find *t* such that *path[t].time >= t - trailTime* and *path[t-1].time < t - trailTime*
4. Interpolate between headColor and tailColor according to time of a point for each point in *path[t..h]*
5. Return *path[t..h]*

### The visualization loop

1. Assume *t* is current time
2. Evolve current visible paths by *dt*
3. Draw current visible paths
4. *t := t + dt*
5. If *t* is greater than visualization time upper bound, *t :=* visualization time lower bound
6. Go to step 1

Due to the fact that time in the loop above always increases and the path arrays are sorted, the algorithm of evolving paths can be improved. We store last *h* and *t,* then next time we need to return visible part we start looking for updated *h* and *t* indices from the old values.

### Particle movement continuity

*dt* in simulation and *dt* in visualization are not necessarily the same. Usually visualization *dt* is smaller. But you can notice in the algorithm above that head and tail indices jump once a condition on current *t* is met. This results in “jerky” end points of visible part of the path.

We solved this by replacing visible path end points with temporary points. Coordinates of these temporary points are computed as an interpolation of two neighboring points in the path array.

## Residual Problems and Limitations

Having implemented all the aforementioned features and optimizations, there are still problems left. Here is a list of most noticeable limitations:

* OpenGL performance. In the end of the day we tell OpenGL to draw a number of polylines. And OpenGL has limited recourses, so as the number of polylines, or the number of vertices in each polyline grows, the program becomes less responsive.
* Simulation performance. As size of meshes, density of faces, strength of vector field or simulation *dt* grow, simulation time grows accordingly. The complexity of simulation of a single path is not really bounded in the worst case – if the vector field is very strong and it goes in circle, crossing multiple faces. Then it is possible that *t* is hardly incremented at each step (because particle reaches face boundary “too soon”), but consecutive path points are too far to be collapsed into a single point (as was described above).  
  Even though this scary case theoretically exists, we did not encounter it. But we can see significant simulation time increase as field strength grows.
* Paths that end up not on mesh surface. Even though we take into account numerical errors during simulation process, there still are cases we are missing. We do not know what situation causes that, but there are cases in which during steps 3 and 4 of the simulation algorithm we find ourselves in the following state:
  + Next particle location is not inside current face
  + Last segment of particle path does not intersect any of current face’s edges

These are rare cases and we don’t really have a solution for them. So if we detect this situation, simulation of that path stops. Also problematic path count is incremented. Problematic path count is printed to console after every simulation.

* Complex mesh geometry is not supported. We support simple meshes of disk or sphere topology. Every face must be triangular. Every edge must have one or two adjacent faces. We never tested the program on meshes that cross themselves or other unusual meshes.

# Summary

The implemented program is a utility for visualization of vector fields. It has the following properties

* Stable
* Supports loading meshes in object file format
* Supports constant fields and fields that vary in time
* Performs parallel field lines computation
* Visualizes vector fields through particle animation

This makes the program usable for vector field researches, as they can now easily and intuitively visualize a vector field the came up with.