Week 1 Report (6/9/16)

Project Tool Update

Initially, the project tool was not working on my computer. The source code required updated libraries such as glut32 header and dll files. Linker library files also needed to be set through visual studios. Values in the stdafx.h file had to be changed to match my target Windows OS which is Windows 10. I also changed some identifiers to fix some errors the tool was having.

Review and Readings

I reviewed the slides going over vector field analysis from your data analytics class which had content mostly covered over during meetings. I went through additional research papers and lectures slides to further familiarize myself with the information. I have been taking notes of all my readings to keep track of all the concepts and study them better. My interpretation is there exists streamlines on a vector field (to visualize flow) and that the periodic orbit around a fixed point classifies which type of steady flow it is (sink, source, saddle). Each fixed point can also be classified by its eigenvalues from a matrix representation. I will keep note of the equations from the readings and what you showed me that apply the eigenvalues mathematically such as Jacobian analysis. I have not fully read over the math and programming behind it but it is my next objective. A vector field is similar to a graph where each point is assigned a vector. 2D vector fields are defined as a discrete mesh which incorporates grid generation and groups the points as triangles to create something like a wireframe.

Tool exploration

There was an issue when exploring the tool dealing with deleting and adding elements. After deleting an element no further elements can be added. Checking ‘show grass’ and ‘use color’ together crashed the program. I do not know the reason for both of these bugs. A couple of functionality I did not understand while working with the tool was frame control. We went over it during our meeting and I have a better understanding of how to save frames and what the functionality does.

Conclusion

In conclusion, I will continue to explore the different functionality and document accordingly to what they do so that I can replicate. I understand at the surface level of adding and assigning element unto the vector field and the concepts between how the fixed points interact with each other. The next step for me would be to look over the code and understand the specifics to create model pseudocode. I will also look over the different types of cross-platform applications and consider which fits well with my programming style and the project demands.
Week 2 Report (6/16/16)

Pseudocode

Mesh generation is started by creating a quad-grid with user-defined input of columns and rows. From there we save all the points in a list and begin triangulation. Triangles should be consistent throughout the graph. There are two types of triangles: clockwise and counter-clockwise. The clockwise triangle is generated by vertices \( \{i, i+1, i+(Nx+1)\}\) for any \(0 \leq i \leq Ny-1\). The clockwise triangle is generated by vertices \(\{i, i+(NX+1), i+(Nx)\}\). After triangulation we can start storing all the triangles in a list which gives each triangle an index to retrieve from. Given the declaration of triangles and their id’s, we can then allow for users to add and edit singularities onto the mesh.

//Goal: To create a vector field from user-defined input regarding mesh range and singularities.

Mesh graph using paintComponent from java.swing

class Mesh{
    List<Point> pointlist;
    List<Triangle> trianglelist;
    int NX, NY; //user input. User-defined set of points (Nx*Ny sample points)
    x_range, y_range = getMax() – getMin();
    x_int = x_range/(Nx-1) //the distance between two points.
    y_int = y_range/(Ny-1)

    //Generate points with loop
    for(int j = 0; j \leq getMax() ; j+= Ny)
        for(int i = 0; i \leq getMax() ; i+= Nx){
            pointlist.add(new Point(i, j));
        }

    //Generate Triangles
    for(int n =0; n \leq pointlist.length()-1; n++){
        Point temp = pointlist[n];
        if(temp.x != Nx && temp.y != Ny){ //corner points do not form triangles
            //clockwise
            Point B = pointlist[n+1];
            Point C = pointlist[n+(Nx+1)];
            new Triangle(0, temp, B, C );
            trianglelist.add();

            //counter-clockwise
Point B = pointlist[n+Nx];
Point C = pointlist[n+(Nx+1)];
new Triangle(1, temp, B, C);
trianglelist.add();
}}
}

class Point{

    float x, y;
    float vx, vy;
    Point(int x, int y){} // constructor
}

Class Triangle{
    Point a, b, c;
    Int type; // 0-clockwise, 1-counter-clockwise
    Triangle(int type, Point a, Point b, Point c){} // constructor
}

class SingularElement{
    int ID; // numbering each element
    int Triangle_ID; // the triangle that contains the singular element
    double centerx, centery; // center under global frame
    int type; // singular type (0-Null, 1-source, 2-sink, 3-saddle, 4-cwcenter,
    // Matrix can be represented with 2d/3d arrays as long there are math operations.
    Matrix transform_matrix; // transformation matrix for user editing
    Matrix Jacobian; // Jacobian matrix is full rank. Used to calculate the vector divergence and curl (attract, repel, saddle).
    double s; // size, scale, geometric representation
    bool deleted;
}

Conclusion

Storing the points and triangles of the mesh in a list is advantageous over a 2d array because of memory accessing since the connecting vertices are already determined and calculated with the data types. I will begin creating the model implementation using Java Swing interface to represent the mesh generation from user input. The model will show each point with a representative ray. I will make sure to keep track of the different complexities while working on this implementation and for the future.
Week 3 Report (6/23/16)

Model Implementation

I created the model implementation representing the mesh generation. Given an input of the number of Nx and Ny, the program will plot all the sample points. Upon clicking the program, a singularity element is added and vector arrows will appear.

Revision

My vectorization was quadrant based rather than performing basis field summation. This includes using a Jacobian matrix for each singularity to calculate the vectorization. Basis is superior to my quadrant implementation because it not only accounts for the detail vector value of every point but it also includes the weights of multiple singularities upon a point. I will also need to add an interface to keep track of which types of singularities are added to the vector field. After that, I will need finish the basis field summation to include and calculate all elements of the vector field.

Conclusion

The math and formulas within each part of the vector field can be simple to grasp but very difficult to implement thoroughly in a programming language. Thankfully since we go through every calculation firmly and review regularly, I can get a clearer idea of how each mathematical principle can be implemented.
Week 4 Report (6/30/16)

Problems Encountered and Revised

Initially, my vectors were determined with quadrant-based calculations. These vectors were wrong because they misrepresented the vector magnitude and direction. The fix was to use Basis Field Summation which used defined Jacobian matrixes to calculate vector values. However, my matrix multiplication had to be fixed from left to right multiplication.

Singular element coordinates are stored in integer values which presented a big problem for weight and distance calculation. Currently, the coordinates for the representative center and their actual center are misaligned because of precision error from producing a floating point number with an integer. My solution for this is to restructure how my program stored its data type variables. This also includes changing how sample points are initialized and represented.

Weights and Formalization

After finishing up the vectorization, I included weights into my calculation for vector values. The first problem was that the vectors were very short. After our meeting, we found that the distance values were very large which made our inverse weight values small. The distance is measured through pixels so the weight formula had to be manipulated by to account for the extra size. It is still unsure but I may need to change the decay weight depending on how the elements interact with each other. Weights are increased by a hundredfold. Vectors were also formalized to make the visualization simple and helps focus more on the direction of the vectors. This was done by dividing each VX and VY component by its distance from each sample.
Conclusion

There was a problem with my variable naming that made the readability of my code confusing. This led to calculation errors being made. My row and column values were labeled as \(ij\) instead of an intuitive label like \(x,y\). Changes in the `paintComponent` need to be made to account for floating point values. I will continue using the debug tool to more effectively fish out any more calculation errors I may have made.

Week 4 Report (7/7/16)

Problems Encountered

My coordinates are still calculated by pixels based on the window rather than over a normalized unit value. This affects the value of the weights (decay and distance) which also distorts the interaction between all the singularities. I will continue to reference off the original code and ask questions along the way to get this done. Hopefully after changing the ranges to unit values, the weights between the singularities and the summation of the vectors will be appropriate. Additional work will also need to be done to account for storing singularity locations and doing sample point calculations. This will be done with debugging and looking through the tool after converting to unit values for the plot points.

Second, center values of singularities are overrun during summation of vector values. This is needed to differentiate exactly where the singularities are and what they are interacting rather than being overridden by another singularity. Currently, this is noticed when more than 2 elements are added onto the vector field and the flow of the singularities overlap. I will need to preserve zero vectors for the centers of singularities. This can be done by performing test cases over singularity locations and creating zero vectors at their center location.

Conclusion/ Goals for Next Week

Singularity center points were corrected. X and Y coordinates were inverted after calculating vector values and so they needed to be switched back.

Functionality will be added with included interface to reset the vector field. This can be done by resetting vector values and releasing elements stored.

Gaussian function will be applied to the vector field to smooth out and approximate the vector values between the singularities.

I will be working with Java OpenGL (JOGL) to add image based flow visualization to further represent the vector field. I will reference the example C program from online to base off what to code. Texture based flow will make the vector field more interactive because the vectors will be continuous which better visualizes the interaction between singularities.
Week 6 Report (7/14/16)

Getting Started With JOGL

Java OpenGL is very similar to regular OpenGL. Many of the functions are named the same and do what they are needed. Setting up a build environment required downloading all the needed jar files and also its respective natives file. The OpenGL profiles initially presented a problem because my computer was only compatible with GL4 and GL2 profiles initially.

Problems Encountered

I was not able to fully translate the sample C program over. Many of the functions in OpenGL C worked differently than they do in JOGL. The most troubled function was `glTexImage2D` which generates the texture image. This function required a buffer which works differently in Java than in C. In C, a 3D array is automatically converted to 1D but for Java we needed to start with a 1D array, represent it as a 3D array and passing it back in the function argument as a 1D buffer. This shouldn’t be too hard to implement with the example online.

Conclusion

My biggest priority is adding my original Java code to JOGL. This will be done by embedding JOGL into a JPanel. Once I can successfully do that, I can start to use JOGL’s drawing capabilities to correct represent the floating point scale for my grid. I want to get domain normalization correct because it would then correct my code’s problem with multiple elements interacting dysfunctional. OpenGL takes care of drawing differently that Java’s graphic library in that it takes care of the pixel point of where to draw. So the benefits is that you don’t have to worry about pixel location while implementation.
Week 7 Report (7/21/16)

JOGL Implementation

I have implemented the vector field from using the graphics library to JOGL. Now the coordinate values are floats and the range and domain of the field is (-1,1). OpenGL takes care of the drawing by using world coordinates rather than window (pixel) coordinates. Therefore I had to convert the pixel coordinates that were recorded through mouse input to world values. The weight function can now be optimized to save center values better and our issue with multiple elements on the field is resolved.

Some additional tasks I need to clean up are normalizing vectors to refine the visual. This can be done by dividing the distance vector by vector value. Also I will tinker the weight coefficient values to optimize the kernel function to reach all points. I will remember to add on the triangle meshes.

Flow Texturing

The OpenGL texturing has been difficult for me to understand. I will work on several online examples with detailed explanations and get back to you as soon as I can with questions and solutions. I believe the texture implementation on C is initialized different than Java so instead of reviewing the C code it would be best to use a Java example. Ideally, the online examples I have found will provide me with an idea of the initialization and details with texturing so that I can reference how it works in the sample C code. I have found java code from OpenGL redbook example. After getting a texture to appear, the animation for the flow be added. I believe this is already done with patterns being calculated so I will just have to correct switch the buffers and redisplay.

Conclusion

My additional objectives for next week is to begin working on implementing regular elements to my vector field. The specifics that I remember are that these are elements that can rotate and scale based on transformations defined by user input. With the formulas you provided I have a clear picture of calculating the vectors and values. I will get back with you if there are any problems.
Week 8 Report (7/29/16)

Regular and Singular Elements Implementation

Attachment and Separation elements have been implemented with the formula you provided me. These implementations do not include a user input theta angle value in their calculations so they run on the theta value of 0 and point to the right. I will implement the new calculations that allow these elements to rotate according to user input.

I will revisit my implementation for singular elements. The focus will be how I’m manipulating the transformation matrix which then allows me to see the changes on rotation and scaling of the singularities. I will also look into the original code and see how you calculated and managed scaling and rotation. Most of the work will deal with user interface or at least the difficult parts I believe. To narrow down and focus on singularity editing, I will create an interface that edit the transformation matrix manually.

Flow Texturing

I have reviewed the OpenGL texture examples. Texture coordinates refer to the vertex which you desire to paste onto the image. Since our viewport is set to the origin, the texture will be shown on the xy axis. I will now work on the IBFV and continue debugging the code to get a demo running. To get a running example quicker, I will try to run the C code with glut plugins. With a working example, I can develop a concept for how the flow works and how it will mix with the vector field. The difficulty with debugging the IBFV code is that textures are saved onto opengl’s display lists which is on the graphics card. I will be debugging the IBFV sample by narrowing down the display functions and noting what is visualized. The goal is to get the 1st frame of the texture to show so that we can know placing a texture may not be the problem.

Conclusion

Most of the work for the coming week is finishing up the work for implementing regular elements and narrowing down the image based flow texture example to something we can translate our vector field to. I will get back as soon with what you wanted for the textures and hopefully and working example. I believe I will run into problems implementing boxes for the singular elements but I will get back to you with specific questions.
Singular and Regular Element Implementation

The progress with regular elements is that the new formula is included which dealt with angles that allowed rotation. Also to explain the original formula, when the constant c is greater than 0 it is a separation, less than 0 is an attachment element, and as 0 the constant will make it a regular element.

To the right is an attachment element angled at a 35 degree angle. Problems that you can see with this implementation is that some of the vector pattern gets overdone away from the element origin. This is due to values dealing with weights and some constants. A little tinkering to get the visuals right is all that is needed.

\[
V(x,y) = B(x,y) \left( \frac{1}{c(y-y_0)} \right)
\]

where \(B(x,y) = e^{-(x-x_0)^2+(y-y_0)^2}\) is the blending function for the element and \(c < 0\) is a parameter that describes the speed.

Original Formula with assumed angle 0 degrees.

\[
V(x,y) = B(x,y) \left( \frac{\cos \theta_i}{\sin \theta_i} \right) + cP(x,y) \left( \frac{-\sin \theta_i}{\cos \theta_i} \right)
\]

where \(P(x,y) = -\sin \theta_i(x-x_0) + \cos \theta_i(y-y_0)\) is the signed

Updated formula with user input theta.

The angle is calculated depending on user’s initial click and location of release. However, the vector values between the multiple elements overlap each other due to the weights being too small. This causes difficulty to differentiate where and the number of elements. The new weight formula involves some constant over \(dx^2+dy^2\). Also, I will have to check not only how attachment elements interact with itself but also with singularities and regular elements by further looking into each of the weight values.

There was a problem with the rotation of singular elements. Upon review, we found that the source Jacobian matrix was being manipulated when an angle was set. However since the Jacobian matrix of source is the identity matrix, there should not have been any change in the calculations and so source and sink elements should not be affected by rotation. Therefore the problem has been narrowed down to my matrix multiplication and calculations. After I can get the rotation to work, I will implement user interface that allows the user to rotate the singularity by dragging rotation points.

Texturing

I tested the image based flow visualization demo in C++ with the original code and it worked as it should. I tried tinkering with the code and commenting out parts to see what did what. The initial drawing with the quadstrips is responsible for the warping animation and the second drawing processing with the blending technique and displaylists works with the black and white texturing. I tried to do the same in JOGL but sadly no success. The texture that is stored in the displaylist seems to not work the same. I believe it is the way JOGL works with buffered images and bytebuffers. That means that textures are encoded differently and must be formatted to accommodate to be accepted in OpenGL. Another difference is that JOGL doesn’t have the same direct access that C does with pointers. This makes binding
and setting textures a little difficult. I will try to put a little more bit of time into JOGL and see what I can learn. Until then, I will be researching about LIQ and using it to implement streamlines.

Changes Added

Rotation has been fixed and implemented as intended. A rotation box has been added to represent orientation of the element. The top of the box is colored to differentiate what orientation the element is in. If any of the vertices and dragged on, an angle is calculated and set onto that element. Rotation can be seen with saddle element as it maintains its cross orientation. Attachment and Separated elements look more visually correct due to changes in weight values.

Element Interaction And Weights

Interaction between singular elements have changed after incorporating rotation and scaling matrices. I believe this might have been a problem with normalization and I have fix it. I will to continue to check if there are other things I have missed.
There are many different types of weights in this vector field that determine the interaction between the elements. The different types of weights can affect the influence or size of the element and the interaction and blending between each of the elements. Ideally, the vectors should blend with each other and now show a sharp vector change. Playing around with the numbers should get the visuals fine-tuned. There is still more fine turning needed to match weight interaction between different type of elements to blend better. However, many other factors can change for the vector field and it will make it hard to recalibrate the weight values. So storing user input weight values will allow easier comparisons. The saved weight values will be done through reading and writing save files.

$$B(x)$$ for Attachment/Divergent Elements -

$$\text{weight} = (C \times \text{Math.exp}(-k \times \text{distance} \times \text{distance}))$$;

- C determines the element’s span onto the vector field
- K determines the “sharpness” of the element and how sharp the vectors converge or diverge

Line Integral Convolution

From what I have read, we can start with arrow plots that then help determine flow direction but not the velocity because the field is normalized. A vector field streamline will then be generated from each point of the flow arrows. This process is called integration and it creates the represented streamline. By stepping forward and backwards from the differential, we can draw the line. From there, we will add a white noise texture onto the flow field to generate the pattern that represents a LIC image by using an image kernel technique. The location of the streamline is mapped from a mesh where the texture coordinates can also be plotted. Below is pseudocode for the streamline and integration.

```typescript
interface Differentiable {
    vec_at(x: number, y: number, v?: Vec2): Vec2;
}

interface Integrator {
    // Step size parameter
    public stepSize: number;

    // Differential calculator function
    public diff: Differentiable;

    step(x: number, y: number): Point;
    stepReverse(x: number, y: number): Point;
}
```
class Euler implements Integrator {
  public v: Vec2;

  constructor(public stepSize: number, public diff: Differentiable) {
    this.v = new Vec2(0, 0);
  }

  step(x: number, y: number): Point {
    this.v = this.diff.vec_at(x, y, this.v);
    return new Point(x + this.v.x, y + this.v.y);
  }

  stepReverse(x: number, y: number): Point {
    this.v = this.diff.vec_at(x, y, this.v);
    return new Point(x - this.v.x, y - this.v.y);
  }
}
Abstract
The objective for this project was to apply the information presented by creating a prototype of the original project but re-implemented on a different language and framework that replicated the visual animation. By creating a prototype and re-exploring the fundamentals of previously established research, models are drawn from the stage of development. This degree of analysis grants the opportunity to study and draw new ideas on top of the previous research to push and branch out the groundwork further. A comparison can be performed on vector field visualization to other visualization techniques in terms of implementation and effectiveness. With a broader understanding of vector fields, different levels of understanding can be worked to use these visualization techniques and models to its full effectiveness.

Introduction
With the implementation of a vector field, additional visual features can be incorporated such as integrating streamlines and creating textures over the field that simulate flow and the interactions between the elements. Despite vector field’s inability to account for increasing complexity of data, the simplicity of vector visualization allows an opportunity to continuously develop and analyze the data through an interpreted design.

Tool Exploration
Working with and understanding the original design allowed me to map out all concepts of what the research wanted to represent. The expansive functionality dealt with multiple fields of visual designs. The code also allowed me to model my own code. The data structuring and program design allowed me to grasp how my model would function.

Vector Field Generation
Generating the vector field started with an understanding of mesh generation. The program needed to have flexibility with how the user wants to represent their visualization and so the mesh was created starting with user-defined input of sample points that determine the graph columns and rows. Through each sample point, the vertices of clockwise and counter-clockwise triangles could be generated and stored. This would be the starting data structure and what we would base our vector field representation on. The clockwise triangle is generated by vertices \( \{i, i+1, i+(NX+1)\} \) for any \( i \): \( 0 \leq i \leq Ny-1 \). The clockwise triangle is generated by vertices \( \{i, i+(NX+1), i+(Nx)\} \). After triangulation we can start storing all the triangles in a list which gives each triangle an index to retrieve from. We would then give each triangle and id and store it in a list for retrieval later. Each triangle and its vertices will be used to store important information such as vector values.

Figure 1: Different design elements showing source (left), sink (middle), saddle (right).
To recreate vector behavior with singular elements, Basis Field Summation is applied which deals with defined Jacobian matrix that determines the type for each singularity. The pattern types we can produce are source, sink, saddle, clockwise center, and counter clockwise center. Each singularity will store a Jacobian matrix that can be manipulated through user interface to change its orientation and scale.

Regular elements that converge and diverge also follow their own formula. They follow the same scheme and structure of singular elements but instead of stored matrices, regular elements change their interaction depending on the value of their determined constant value.

Regular elements change their interaction depending on the value of their determined constant value.

Original Formula with assumed angle 0 degrees.

\[ V(x, y) = B(x, y) \left( \frac{1}{c(y-y_0)} \right) \]  \hspace{1cm} (5)

where \( B(x, y) = e^{-(x-x_0)^2+(y-y_0)^2} \) is the blending function for the element and \( c < 0 \) is a parameter that describes the speed.

Updated formula with user input theta.

\[ V(x, y) = B(x, y) \left( \frac{\cos \theta_0}{\sin \theta_0} \right) + cP(x, y) \left( \frac{-\sin \theta_0}{\cos \theta_0} \right) \]  \hspace{1cm} (6)

where \( P(x, y) = -\sin \theta_0 (x-x_0) + \cos \theta_0 (y-y_0) \) is the signed.

Figure 2: Mathematical formula for regular elements.

Storage of all elements will be in each of their respective list so rendering all the elements would simply require iterating those lists and plotting the elements during each display. The interaction between the each of the elements are determined through weight values that are each calculated differently for their respective element. The base weight equation is weight=\( C \cdot e^{(-k \cdot distance \cdot distance)} \) where \( C \) is a constant determining element span and \( K \) is a constant determining element sharpness. Vector values are the summed values of all the elements and so the finalized value will visually present the merging of multiple elements.

Rotation and Scaling of elements

Singular elements had the functionality of rotation and scaling by adding transformation matrices into their calculations which creates a new Jacobian matrix with manipulated values. These transformation include the rotation matrix and the scale matrix. An updated formula was used for regular elements for rotation which incorporated a user designed angle value.

User interface was also implemented to rotation to user specification. This was done by either having rotatable boxes around elements. Additionally, users could also select elements with a selection mode to modify weight values and their scaling.

Choosing our Platform

Java is a common platform for running on many hardware types. Java applications also have the ability to run on systems given that they have a JVM. The original flow based program which was reference to begin this project was written in C++. However, the application was unable to work on a different operating system than the initial OS that it was developed in until some tinkering was done. The MFC libraries within the C++ programs faced inconsistencies when developing on a different type of supported OS. Changes in the Windows header of standard system include files were done to run the application and would need to continuous be done when switching operating systems.

We chose Java OpenGL for our visual platform since it was easier to reference and work off the existing use of the OpenGL’s libraries.

A given the task for this project was translating an existing C program that sampled the flow visualization texturing using OpenGL. However, due to my inexperience and the differences between
JOGL and OpenGL, I could not replicate the same visualization behavior in my prototype using JOGL. A static texture image would be generated and stored onto the graphics card using OpenGL’s displaylists and then displayed over. A simpler texturing design was turned to which was Line Integral Convolution (LIC). LIC is CPU based rather than working on top of the graphics card. It dealt more with manipulating images which was simpler to implement.

JOGL drawing capabilities are handled by embedding the drawing canvas onto a JPanel. The drawing domain of JOGL’s canvas is different from how swing graphics is since the swing graphics draw on a pixel-based graph and its domain is dependent on the panel’s size. JOGL represents their drawing domain in a floating point scale which uses world coordinates rather than window (pixel) coordinates. This difference changes the needed sizes of the weight values of the elements and the interaction that was observed between the elements. However, since Java’s mouselistener receives windows coordinates, those given window coordinates had to be converted to world coordinates.

```java
float x = 2.0f * mouse.getX() / canvas.width - 1.0f;
float y = -2.0f * mouse.getY() / canvas.height + 1.0f;
```

**Figure 3:** Pseudocode for converting window (pixel) coordinates to world coordinates in OpenGL.

**Visualization**

Representation of the flow of the field is delivered through vector arrows, streamlines, and line integral convolution textures. When drawing the arrows, OpenGL can render the arrows using starting and ending position. The program iterates through all sample points and uses the location of those sample points to plot the vector arrows. However, to reduce visual clutter all vectors are normalized by a specific length value to evenly create a visual field representation.

**Figure 4:** Singular element interaction (left) with rotation boxes and Regular element interaction (right).

Streamlines are the lines tangent to the vectors at all points. To start calculating streamlines, there needs to be a way to integrate the differential function of the field. Vector values will need to be calculated at any given point instead of just at the values stored at the sample points. Since the vector values are stored only at the vertices of the triangle mesh, barycentric interpolation will be used which uses the relativeness
of the coordinates to each vertices to calculate the vector value. We start with Euler integration for an example representation and then apply Runge-Kutta integration for precise streamline values.

\[
\begin{align*}
  k_1 &= \frac{h}{f(x_n, y_n)} \\
  k_2 &= \frac{h}{f(x_n + h/2, y_n + k_1/2)} \\
  k_3 &= \frac{h}{f(x_n - h/2, y_n - k_2/2)} \\
  k_4 &= \frac{h}{f(x_n + h, y_n + k_3)} \\
  y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}
\end{align*}
\]

Figure 5: (a) RK4 Integration used to draw streamlines and (b) Barycentric coordinates to find relative vector values.

Streamlines are also drawn onto the field using the vector values of each point. At a determined point around the center of an element we perform RK4 integration to create an accurate estimate of a slope for the field and then take a step and repeat till our streamline is finished drawing. LIC allows to better visually represent the flow of the field and it is down by creating a white noise and convoluting it with a streamline. This convolution is done by recording the pixel values across the streamline, summing up each of their color values and dividing the total by the number of pixels. The final color should be gray and show a contrast between the pixels located around the streamline. The final LIC image should be stored in a 2D array so that it can be used by OpenGL to paste the image on the field.

Figure 6: Representing streamline using Euler (left) and streamlines convoluted with white noise to show LIC (right).

Conclusion
Incorporating and applying pre-established visualization techniques presented an extraordinary learning in the field of computer graphics. The prototype developed was made to implement an existing design for vector fields and examine the concepts and fundamentals of flow visualization. Although many of these visualization topics have been explored, this project served to reiterate and identify areas of the development which can be optimized. Getting started with this research project required thorough reading of research papers which allowed an opportunity to be familiarized with the mathematical analysis of vector fields and the way they modeled different effects.