2013

Streamsurface Smoke Effect for Visualizing Dragon Fly CFD Data in Modern OpenGL with an Emphasis on High Performance

Jordan Sipes
Wright State University

Streamsurface Smoke Effect for Visualizing Dragon Fly CFD Data in Modern OpenGL with an Emphasis on High Performance

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Jordan D. Sipes
B.S.M.E., Cedarville University, 2004

2013
Wright State University

Wright State University
SCHOOL OF GRADUATE STUDIES

April 3, 2013

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Jordan D. Sipes ENTITLED Streamsurface Smoke Effect for Visualizing Dragon Fly CFD Data in Modern OpenGL with an Emphasis on High Performance BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Thomas Wischgoll, Ph.D.
Thesis Director

Mateen M. Rizki, Ph.D.
Chair, Department of Computer Science and Computer Engineering

Committee on
Final Examination

Thomas Wischgoll, Ph.D.

Yong Pei, Ph.D.

Michael L. Raymer, Ph.D.

R. William Ayres, Ph.D.
Dean, School of Graduate Studies

ABSTRACT

Sipes, Jordan. M.S.C.E., Department of Computer Science and Engineering, Wright State University, 2013. *Streamsurface Smoke Effect for Visualizing Dragon Fly CFD Data in Modern OpenGL with an Emphasis on High Performance.*

Visualizing 3D, time dependent velocity vector fields is a difficult topic. Streamlines can be used to visualize 3D vector fields. A smoke effect where the streamline is faded out as time progresses can provide a better visualization of a time dependent flow. This work uses modern OpenGL to create a smoke trail effect with streamsurfaces in the dragon fly data set. Many aspects affecting performance are tested to determine the best options or approach.

Contents

List of Figures

List of Tables

Dedicated to

My Parents

Introduction

The goal of this project is to create a visualization affect that simulates smoke in a time dependent fluid flow. CFD data for a flapping deformable wing is used. OpenGL 3.3 is used for real time rendering. The software uses C++ and Qt so it can run on Linux, Windows, and theoretically Mac OS X.

Related Work

2.1 Flow Visualization Background

CFD data typically consists of a velocity vector field in either 2D or 3D, and can be time dependent or static. There are several common approaches to visualizing flow data, the simplest being vector glyphs [1]. With vector glyphs, an arrow or line is drawn at each data point indicating the direction and magnitude of the flow. This method works for simple 2D flow visualizations but this method is very cluttered when 3D flow is visualized with it.

Another method for visualizing flow data, streamlines, is less cluttered than the vector glyph method. Streamlines are traces of a mass less particle advected through the flow [1]. The result is a line tracing the path of each particle through the flow. If this method is used for time dependent flow the streamlines are referred to as streaklines. An extension of streamlines are streamsurfaces. A streamsurface is made up of a line of streamlines connected together to form a surface. This project focuses on the use of streamsurfaces in time dependent flow.

A third method of visualizing flow data is texture based methods such as line integral convolution or LIC [1]. In order to create an LIC visualization a noise texture is smeared in the direction of the flow at each point. This method works well for 2D flow but is difficult to adapt to 3D flow.

2.2 Related Works in Flow Visualization

An early attempt at visualizing 3D unsteady flow was done by Bryson and Levit in their 1991 work on a virtual wind tunnel [2]. They created an entire virtual environment for interactively investigating an unsteady 3D flow. Specifically, Bryson and Levit wrote software to display streamers, streamlines, streamlines, and basic material tracking in real time. What is more interesting is their visualization system used a boom mounted monitor, head tracking, and a data glove for interactive visualization. The computer they used for the visualization is an SGI 380 VGX with only 48 MB of RAM and eight 33 MHz CPUs. Bryson and Levit were able to achieve some real time visualization with this system. They discuss memory and computational limitations with the possibility of using striped RAID disks in the future.

An early work on streamsurfaces was done by Hultquist in 1992 [3]. Hultquist created stream surfaces by linking streamlines together in to ribbons with triangles. His algorithm parametrized the problem according to streamlines and time lines. This method was successful at the time of writing.

Cabral and Leedom originally created the line integral convolution (LIC) method [4]. LIC starts with a noise texture that is blurred locally along the vector field. This method produces a good overall view of the flow field. The algorithm also lends itself well to parallelism.

Shen and Kao demonstrated a modification to the line integral convolution algorithm for unsteady flow that they refer to as UFLIC (unsteady flow line integral convolution) [5]. This method uses value depositing and a successive feed forward algorithm to create an accurate representation of the flow. This approach is primarily useful for 2D or surface flow visualization. Shen and Kao's algorithm also has issues with aliasing affects and being blurry when velocity vectors change direction rapidly.

Verma, Kao, and Pang merged streamlines and line integral convolution (LIC) to create a visualization affect referred to as PLIC (pseudo line integral convolution) [6]. Their

method tried to get around the issue of seed placement commonly associated with streamlines. The PLIC enhancement Verma, Kao, and Pang propose to LIC is done by first generating an LIC image, then tracing streamlines and picking pixels for the streamlines based on the pixels in the LIC image. These pixels are added for all streamlines traced. The result is a cleaner flow visualization that is weighted with information from both methods.

Laramee et. al. compared different methods of visualizing 3D CFD swirl flow [7]. Their work primarily concerned steady state flow, and discussed the positives and negatives of direct, geometric and texture based visualization for this type of flow. For direct flow visualization they evaluated using colors and arrow glyphs. Their geometric flow work involved streamlines and their texture based approaches center around LIC. They concluded that direct approaches get cluttered in large 3D simulations. Geometric visualization can tell quite a bit about the internal structure of a flow but this method is highly dependent on picking the correct seed points. Laramee et. al. determined that texture based approaches are good for viewing what is going on overall, but are difficult to use in 3D.

Schafhitzel et. al. proposed a point based streamsurface visualization algorithm [8]. Their method was able to achieve interactive, real time rendering through the use of parallel computations on the GPU. Schafhitzel et. al. also incorporated LIC into a 3D streamsurface visualization. With regard to their GPU implementation, flow data is packed into a 3D texture. The particles are also stored in a 2D texture. This allows all of the calculations to take place in the graphics card through the use of shaders. This approach was also used for Schafhitzel et. al. LIC visualization. Unfortunately, the code was designed for DirectX 9.0 using HLSL and Shader Model 3.0, and thus limited to Microsoft Windows. The end result of Schafhitzel et. al. work was a LIC texture mapped to a streamsurface in real time.

An interesting solution to topology problems with streamsurfaces was presented by Schneider et. al. in their work on Topoloy Aware Streamsurfaces [9]. Since streamsurfaces are a series of streamlines connected to form a surface, problems arise when divergent flow is encountered, particularly saddles. In order to detect problem areas they calculated

the Jacobian matrix along the streamsurface and it's eigenvalues. When highly divergent flow was encountered the streamsurface was split. This reduced the number of streamlines needed to visualize the flow and therefore reduced the computational complexity.

In their work on image space advection on graphics hardware Garber and Laramee [10] performed image space advection entirely on the GPU. Image space advection calculates the flow direction of an unsteady 3D on the surface of a body. What makes this work interesting is they did this in real time by using shaders and textures in the graphics card. The information was not passed back to the CPU for any calculations, thus, the running speed was quite high. They were able to achieve real time frame rates on common graphics hardware in 2004.

Camp, et. al. proposed solutions for parallel computation in a distributed-memory system of streamsurfaces for large data sets [11]. Since the main issue lies with new streamlines having to be added as the stream surface is calculated, they explored the performance of different parallel partitioning schemes. These partitioning schemes can be characterized in to two different categories, parallelize over particle (POP) and parallelize over data (POD). POP works by giving each parallel task a fixed amount of points to start with. The big advantage of POP is a negligible amount of cross task communication is required. On the other hand, the downside of POP is a high I/O cost. In order to get around the I/O cost, POD partitions the computational tasks around the data. Consequently, POD has performance issues with imbalance. The results from Camp, et. al.'s work show the parallelize over points method produced the lowest running time in most cases for parallel streamsurface calculations.

Use of the accumulation buffer for anti-aliasing was described by Haeberli and Akeley in their 1990 work [12]. They rendered objects multiple times with a fixed amount of offset in different directions and stored the results in the accumulation buffer. Haeberli and Akeley demonstrated a Gaussian filter, when applied to this anti-aliasing method, provided the best results visually. They also discussed how the accumulation buffer can be used for

motion blur. The limitation of this approach is the objects had to be drawn in the correct z-buffer depth to get the correct affect. Haeberli and Akeley also show how the accumulation buffer can be used for a depth of field affect and a soft shadow affect by using multiple rendering passes to the accumulation buffer.

Winner et. al. discussed anti-aliasing methods using the accumulation buffer that solved some problems previously associated with accumulation buffer anti-aliasing algorithms, namely depth sorting in hardware [13]. They achieved this with hardware that supported a z-buffer for depth sorting. Winner et. al. were able to achieve anti-aliasing of a full scene for only a 30 percent performance penalty.

Gribel et. al. described a motion blur approach that was accurate and high performance [14]. They used a per-pixel lossy compression algorithm to reduce the memory cost. With the use of DirectX 11 and floating point frame buffers Gribel et. al. were able to achieve smooth motion blur without sampling artifacts. Their algorithm relied on CPU calculations that made it not possible to achieve real time rendering.

A smoke visualization affect algorithm was demonstrated by Angelidis et. al. in their work on a vortex based smoke simulation [15]. Their method used a filament similar to a ring that is artificially created and deformed according to the flow vectors. Points were then seeded around the filament and advected through the flow. The filament was also advected and stretched through the flow. Principle component analysis was used on the filament to vary the intensities of particles around it. They were able to accelerate parts of this algorithm on the GPU using OpenGL 2.0 pixel buffer objects and the floating point render target extensions.

Data

The visualization software written for this project targets the dragon fly deformable flapping wing data developed by Christopher Koehler for his PhD. Thesis [16]. High speed cameras were used to film dragon flies taking off from the ground. Wing positions were recorded from the camera data by placing dots on the dragon fly wings. The position data was used to create a CFD simulation of the air flow using Tecplot. The Tecplot data files are what this project is trying to visualize.

3.1 Tecplot Velocity Data

The dragon fly data was in a Tecplot format initially. It was difficult to find any information pertaining to the layout of the Tecplot files. One was opened in a text editor and it was discovered that the data is in ASCII format in space delimited columns. These files are very large, each time interval takes up 1.4GB of space in this format. For a real time simulation, a faster, more compact format is needed. Also, judging by the column headers in the Tecplot file, there is a lot of information in there that is not needed for this visualization. A conversion program was written to convert the Tecplot files to the VTK legacy version 2.0 binary format [17] since it is commonly used in visualization and provides a nice compact storage format. The conversion program also strips out everything except the velocity and position data.

Paraview is a visualization program build around the VTK library by Kitware. As a

result, this is a good standard of comparison. Paraview was used to check the validity of the binary VTK files generated by the conversion program. One difficult to solve problem was the data in the VTK binary files needs to be in big endian format, this was not very clear in the documentation. In order to figure this out a known ASCII VTK file was converted to binary in Paraview. A hex editor was used to examine the file and the endianness issue was discovered. Converting the data to the binary format reduced it to 174MB per time slice. It should be noted that the data is in a structured grid VTK format. The grid doesn't change position over time, so the position data is loaded just once, and the offset for the velocity data is determined when the first file is loaded making loading subsequent files significantly faster. This still causes a lot of memory to be used, approximately 86MB per time step. Considering there are 800 time steps available to render not all can be loaded at once. A lot of the testing was done on a machine with 16GB of RAM. The visualization program was able to load 162 time steps at once on this machine and still have a few hundred MB of RAM left over. This testing was in openSUSE Linux running the KDE desktop. The interesting part of the data is mostly in the center, so it might have been possible to extract a block of data from the center for each time interval and just use it for the visualization. This is apparent in Figure 3.1.

3.2 Wing Data

In order to have a decent looking visualization the wing models are needed. The wing position data is in the Tecplot files but was hard to locate. Through trial and error it was found that one of the columns in the Tecplot file named 'ghost' contains the wing data. If the value in the ghost column is a 1 then the cell is wing cell, other wise it is a 0. A program was created that dumped each column and tried to render it several different ways. This is how the wing position data was discovered. This information simply tells if the CFD mesh coordinates lie inside a wing or not. Some problems that result from this data format are

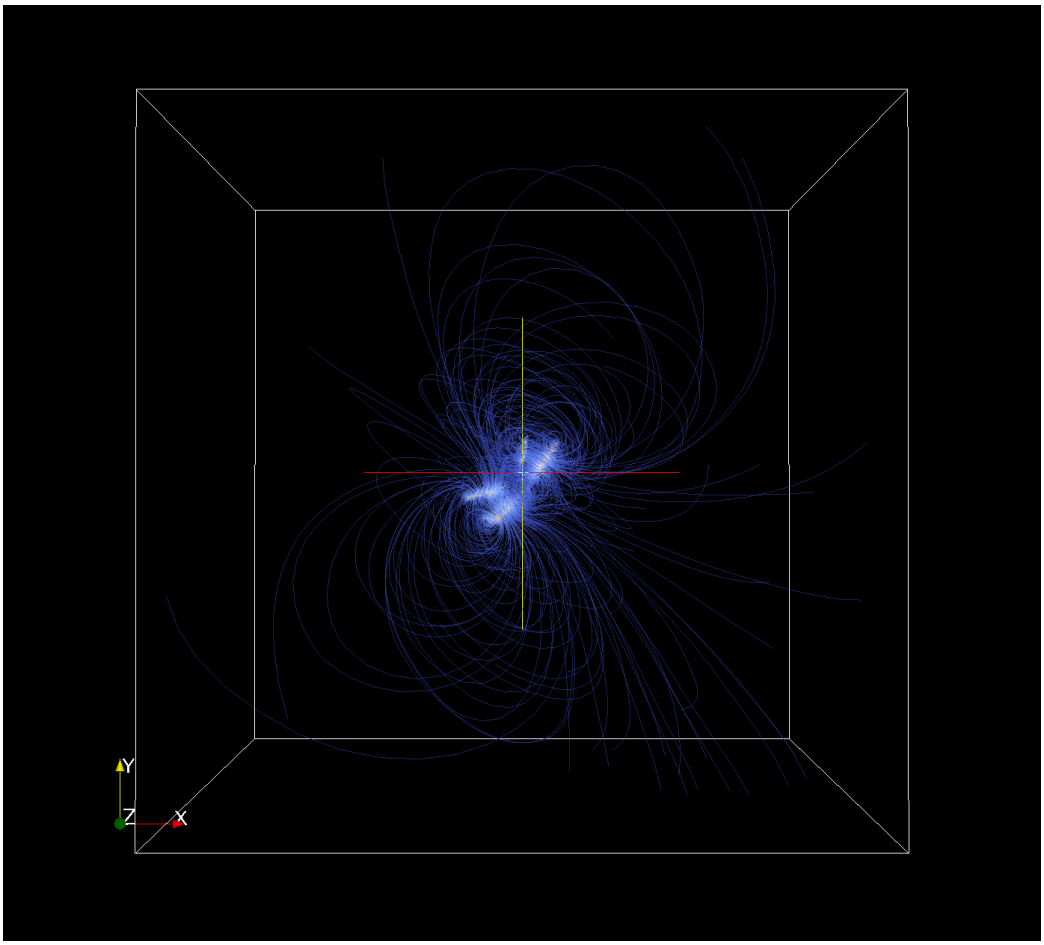

Figure 3.1: Data Mesh Size Viewed in Paraview

the data is very coarse for the wing position and it is not connected in a mesh of any sort. This is shown in Figure 3.2. Here Paraview is used to draw spheres at each wing data point. The visual quality of the resulting representation is clearly insufficient.

In order to draw the wings triangles defining the outer surface are needed. To convert what is effectively a point cloud to a surface Paraview was used. The point cloud was treated as if it were an isosurface. An isosurface is similar to a contour line on a map that shows a line through a constant elevation but in this case it is a surface in three dimensions. The marching cubes algorithm can be used to separate a surface based on a thresh hold value [1]. Paraview has the ability to separate isosurfaces. As mentioned previously, a program was written to extract the wing position data from the Tecplot files and put it in a

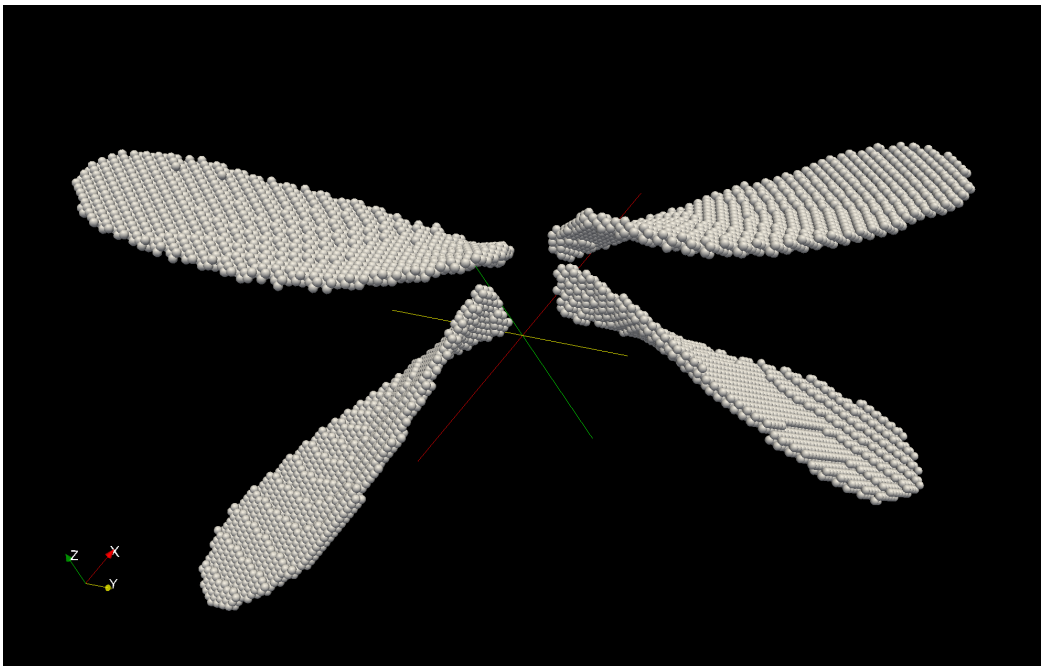

Figure 3.2: Initial Wing Data Rendered in Paraview

VTK file as scalar data. Loading these files into Paraview and creating an isosurface with a threshold of 0.5 results in a crude representation of the wing. Paraview can export the models to a 3D mesh in a PLY format. This format is easy to load with other software. All of this was scripted in Paraview using the Python console, since there were 800 files to convert. The default Paraview package in openSUSE does not have Python enabled, so the source RPM was obtained, the code was patched to allow the Python console to build, and the package was rebuilt. Another solution would have been to download the prebuilt version from Paraview's website, but this version was compiled on RHEL 5 with an old version of GCC. Since quite a few files needed to have the wing surface extracted the rebuild of openSUSE source RPM was more desirable for performance reasons.

The resulting mesh is very coarse, shown in Figure 3.3. To alleviate this problem the PLY exports were loaded into Blender. Vertex smoothing was used with 4 passes yielding the best results. Normals were added by Blender by enabling smooth shading. The meshes were then exported in the PLY format. The smooth result is shown in Figure 3.4 All of this was scripted using Python due to the large file count. One major problem with this

approach is Paraview caused some of the wings to look like they stuck together during the isosurface generation. These had to be fixed manually in Blender.

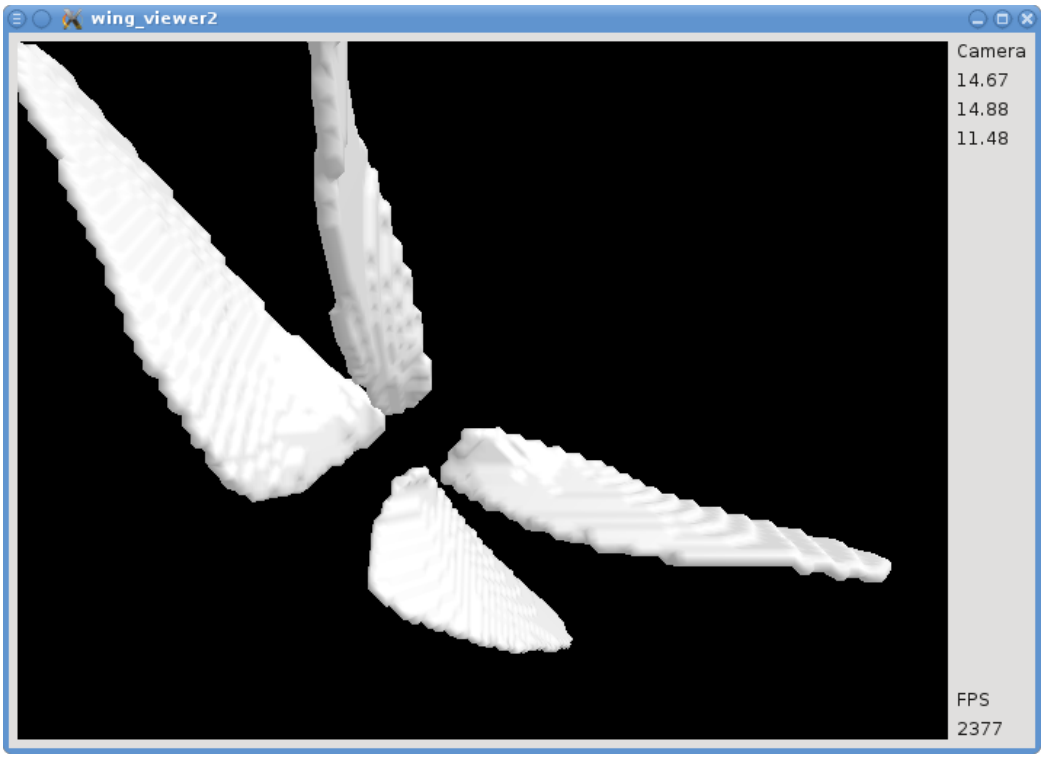

Figure 3.3: Wing Before Vertex Smoothing in Blender.

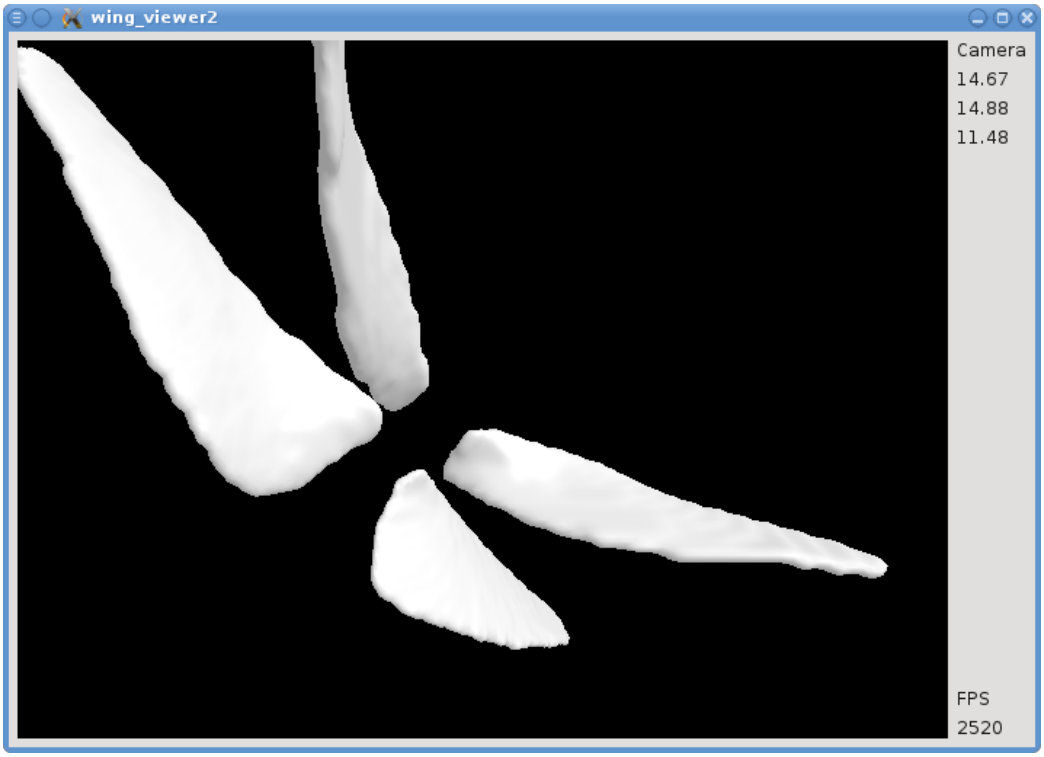

Figure 3.4: Wing After Four Passes of Vertex Smoothing in Blender

Numerical Integration

The goal of this visualization method is to provide a real time smoke trail affect. To achieve this, stream lines need to be traced through the velocity vector field. Since raw velocity data is only available a numerical integration approach is necessitated. A 4th order Runge-Kutta method was chosen for both speed and accuracy.

4.1 Euler Method

The Runge-Kutta method is an extension of the Euler method of integration [18] [19] shown in Equation 4.1.

$$y_{i+1} = y_i + f(x_i, y_i)\, h \tag{4.1}$$

From this equation a new position can be found by multiplying the derivative $f(x_i, y_i)$ by the interval h. Equation 4.1 can be used for one dimensional integration but the data set that is being modeled for this project is a velocity vector field in 3D. So, to apply equation 4.1 to the vector field we multiply the velocity vector by some time step to get a new position in 3D space. The only problem with this approach is the Euler method is not very accurate. Consequently, to solve this the 4th order Runge–Kutta method is used.

4.2 4th Order Runge-Kutta Method

Various attempts were made to extend the Euler method to add accuracy [18]. On of the more successful methods that resulted from this is the 4th order Runge-Kutta method. The 4th order Runge-Kutta method extends the Euler method by adding several intermediate steps by which the order number of the method is named. For the 4th order method, the equation is 4.2.

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)h \tag{4.2}$$

where

$$k_1 = f\left(x_i, y_i\right) \tag{4.3}$$

$$k_2 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right) \tag{4.4}$$

$$k_3 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h\right) \tag{4.5}$$

$$k_4 = f\left(x_i + h, y_i + k_3 h\right) \tag{4.6}$$

Intermediate steps k_1, k_2, k_3, and k_4 need to be calculated first, in numerical order. Then, the results are substituted back into Equation 4.2 to get the next value.

In order to use Equation 4.2 in the CFD flow data the velocity for the current location is looked up first. Since this position will almost never be right on a data point, interpolation is needed. Therefore, trilinear interpolation was used to find the velocities in the vector field. Trilinear interpolation is simply a weighted average in one dimension, then in another, and finally in the last. A LaGrange interpolation method was explored also, but this took considerably more time to execute and was not needed. It should be noted that each $f\left(x_i, y_i\right)$ evaluation causes a look up of the velocity in the data. These look ups are more expensive than the floating point math and, consequently, the primary reason a 5th order method was not used.

4.3 Higher Order Runge Kutta Methods

Several higher order integration methods were explored for this work, the first being a fifth order method called Runge Kutta Fehlberg or RK45. This method uses coefficients organized in such a way that only six look ups are required to get a 5th order solution, and you get a 4th order solution without any more look ups [20]. Having both 4th and 5th order information at no extra cost is useful for calculating the error and adjusting the step size. For this project, adjusting the step size is not an option since the goal is real time rendering. A 4th order method was used because it is faster than any of the 5th order methods. Nevertheless, the accuracy of this 5th order approach was compared to the 4th order to validate it's accuracy for this problem. The Runge Kutta Fehlberg equations are shown next. 4th order:

$$y_{i+1} = y_i + \left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right)h \tag{4.7}$$

5th order:

$$y_{i+1} = y_i + \left(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 +\right)h \tag{4.8}$$

where:

$$k_1 = f\left(x_i, y_i\right) \tag{4.9}$$

$$k_2 = f\left(x_i + \frac{1}{4}h, y_i + \frac{1}{4}k_1 h\right) \tag{4.10}$$

$$k_3 = f\left(x_i + \frac{3}{8}h, y_i + \frac{3}{32}k_1 h + \frac{9}{32}k_2 h\right) \tag{4.11}$$

$$k_4 = f\left(x_i + \frac{12}{13}h, y_i + \frac{1932}{2197}k_1 h - \frac{7200}{2197}k_2 h + \frac{7296}{2197}k_3 h\right) \tag{4.12}$$

$$k_5 = f\left(x_i + h, y_i + \frac{439}{216}k_1 h - 8k_2 h + \frac{3680}{513}k_3 h - \frac{845}{4104}k_4 h\right) \tag{4.13}$$

$$k_6 = f\left(x_i + \frac{1}{2}h, y_i - \frac{8}{27}k_1 h + 2k_2 h - \frac{3544}{2565}k_3 h + \frac{1859}{4104}k_4 h - \frac{11}{40}k_5 h\right) \tag{4.14}$$

Another Runge Kutta integration method is the $Cash - Karp$ method[21]. The $Cash - Karp$ approach has the option of choosing the order based on the error which can greatly save computation time. Unfortunately, the 4th and 5th order calculations both require the same number of look ups, so the running time should be similar. The $Cash - Karp$ method was compared to the Fehlberg and 4th order methods and the equations are as follows. 4th order:

$$y_{i+1} = y_i + \left(\frac{37}{378}k_1 + \frac{250}{621}k_3 + \frac{125}{594}k_4 + \frac{512}{1771}k_6\right) h \tag{4.15}$$

5th order:

$$y_{i+1} = y_i + \left(\frac{2825}{27648}k_1 + \frac{18575}{48384}k_3 + \frac{13525}{55296}k_4 + \frac{277}{14336}k_5 + \frac{1}{4}k_6 +\right) h \tag{4.16}$$

where:

$$k_1 = f\left(x_i, y_i\right) \tag{4.17}$$

$$k_2 = f\left(x_i + \frac{1}{5}h, y_i + \frac{1}{5}k_1 h\right) \tag{4.18}$$

$$k_3 = f\left(x_i + \frac{3}{10}h, y_i + \frac{3}{40}k_1 h + \frac{9}{40}k_2 h\right) \tag{4.19}$$

$$k_4 = f\left(x_i + \frac{3}{5}h, y_i + \frac{3}{10}k_1 h - \frac{9}{10}k_2 h + \frac{6}{5}k_3 h\right) \tag{4.20}$$

$$k_5 = f\left(x_i + h, y_i - \frac{11}{54}k_1 h + \frac{5}{2}k_2 h - \frac{70}{27}k_3 h + \frac{35}{27}k_4 h\right) \tag{4.21}$$

$$k_6 = f\left(x_i + \frac{7}{8}h, y_i + \frac{1631}{55296}k_1 h + \frac{175}{512}k_2 h + \frac{575}{13824}k_3 h + \frac{44275}{110592}k_4 h + \frac{253}{4096}k_5 h\right) \tag{4.22}$$

4.4 Error Calculations

Calculating the error in the integration routine is very important. There are two common approaches to error estimation for Runge–Kutta integration. The first method is step doubling or halving [19]. For step doubling, the error for the 4th order method varies by the following:

$$\Delta = h^5 \phi \tag{4.23}$$

In Equation 4.23, ϕ is for the step [19]. Thus, the error is directly related to the step size. Therefore, by varying the step size, such as doubling it, and comparing the result, a good idea how bad the error is can be ascertained. This comparison is shown in the following equation [19]:

$$\Delta = y_2 - y_1 \tag{4.24}$$

The other method for comparing error that was explored for this project involves using a lower order method and comparing the difference. This comparison is what the Fehlberg and Cash-Karp methods use[21][20]. For testing in this work the step halving error calculation was used since it is easier to do with the 4th order method. All other 5th order methods used the 5th calculation and consequently took more time to execute.

Graphics

As mentioned previously, the focus of this work is to create an efficient visualization of CFD flow data using a smoke trail affect. In order to create a smoke trail affect a stream-surface is created and faded out as time passes. Concerning the GPU implementation, the OpenGL accumulation buffer can be used or frame buffer objects and OpenGL shaders. For this work the OpenGL accumulation buffer was used as a base line comparison for performance.

5.1 OpenGL Library

OpenGL is a cross platform library that gives access to 3D functions on a graphics hardware. The OpenGL library specification is an open specification that is maintained by the Khronos group [22] who control what features are added. Different versions of the library add or remove features. Consequently, the minimum version that is required for this work is OpenGL 3.3. A newer version should work fine also as long as OpenGL 3.3 features are available. It would also be possible to use only OpenGL 3.0 if certain extensions were available. These extensions are frame buffer objects and floating point textures. Layout qualifiers were used in the shaders but these could be eliminated with changes to the shader loading code.

5.2 OpenGL Accumulation Buffer

OpenGL has several frame buffers that are available to an application [23]. For instance, color buffers contain per pixel color information and are what an application draws to most of the time. Some common color buffers for rendering are the front and back buffer. Another type of buffer is the depth buffer which is used to determine what objects are drawn when depth testing is enabled. Restricting drawing to a certain area can be done with stencil buffers. The accumulation buffer is used to copy one of the color buffers. It is not possible to draw to the accumulation buffer directly, but it is possible to specify a value to multiply all of the colors by. By varying this multiplier a fade affect can be achieved.

In order to use the OpenGL accumulation buffer for a fade affect, the application must first ask the window manager for an accumulation buffer. Next, the scene is then drawn to the default frame buffer which is usally the back buffer assuming double buffering is enabled. After this, what was previously in the accumulation buffer is multiplied by a fade value using the following code. Then, the fade rate is multiplied by the time since the last update to get a consistent fade affect.

```
1 glAccum(GL_MULT, 1.0 - Δ_time * fade_rate);
```

Next, the current draw buffer is copied to the accumulation buffer and added to what is already in the buffer. Finally, the contents of the accumulation buffer are copied back to the screen (back buffer) as shown in the code below.

```
1 // copy to accumulation buffer
2 glAccum(GL_ACCUM, 0.1f);
3 // copy accumulation buffer to back buffer
4 glAccum(GL_RETURN, 1.f);
```

5.3 OpenGL Shaders and Frame Buffer Objects

With the advent of OpenGL 3.0 most of the fixed function pipeline is deprecated [24]. More specifically for this project, the accumulation buffer is deprecated. This does not mean it is removed entirely, but there is no guarantee that it is available or implemented in hardware. Therefore, a different option to the accumulation buffer was explored, specifically, using Frame Buffer Objects and Shaders.

5.3.1 OpenGL Shaders

Programmable shaders were first added to OpenGL in 2004 with the release of version 2.0 [25]. Shaders are simply user programmable stages of the rendering pipeline. This allows for a great deal more control in programs versus the old fixed function pipeline. The following figure 5.1 taken from the OpenGL 3.3 Specification [26] shows the OpenGL pipeline.

Vertex data is first passed in by the application. Following this, in the second stage, also called the geometry stage, the verticies are converted to primitives. If textures are used they are applied in the geometry stage. Next, the geometry data is then passed to the fragment stage. Operations can be executed per fragment in this stage. For this work only the vertex and fragment stages of the pipeline are used. These are the minimum shader stages that have to be implemented by the programmer per the OpenGL specification.

5.3.2 Frame Buffer Objects

Frame buffer objects (FBOs) are very similar to the default render buffers that an application normally draws to but with some extra flexibility [27]. For instance, frame buffer objects can have multiple color buffers attached to them or a texture. Multiple color buffers allow shaders to write to multiple color outputs at once. In addition, by attaching a texture

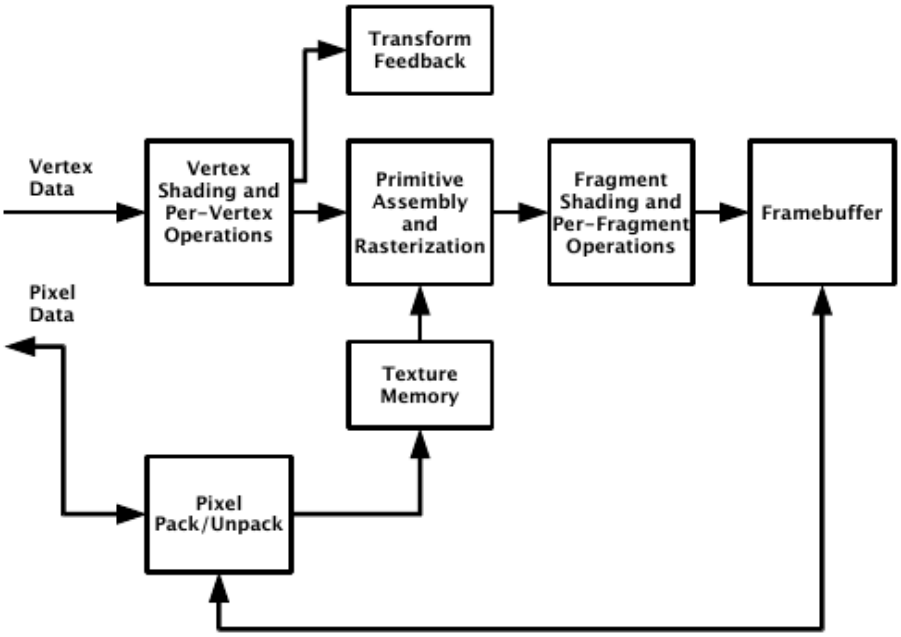

Figure 5.1: OpenGL Pipeline from the Version 3.3 Specification.

to a frame buffer object the application has the ability to render directly to a texture. Incidentally, rendering to a texture is very useful for off screen rendering operations because it is possible to render directly to a floating point texture so that accuracy is preserved. In order to render to a texture, the frame buffer object is first bound, then the scene is rendered, next the default frame buffer is bound, and finally the off screen texture attached to the frame buffer is drawn [28]. The procedure for the fade affect with frame buffer objects is similar to the fade affect with the accumulation buffer.

The render to texture approach is used heavily in this project. First, a texture has to be created for the frame buffer, the following code shows how to do this.

```
1 glGenTextures(1, &fbo_accum_texture);
2 glBindTexture(GL_TEXTURE_2D, fbo_accum_texture);
3 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, ...
      GL_CLAMP_TO_BORDER);
4 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, ...
      GL_CLAMP_TO_BORDER);
5 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
6 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
7 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, fbo_w, fbo_h, 0, GL_RGBA,
8      GL_FLOAT, NULL);
```

The key here is the texture storage format, a floating point storage format is needed for the fade affect to work. Either GL_RGBA32F, GL_RGBA16F, or GL_R11F_G11F_B10F has to be used here. As I will explain in the results section, GL_RGBA16F was much faster than GL_RGBA32F. The GL_R11F_G11F_B10F format did not produce desirable results most likely due to round off error. Once the texture is created the frame buffer is created, bound, and the texture is attached. At this point the frame buffer is read to be used. The code for

the FBO creation, binding, and attachment is shown in the next code listing.

```
1 glGenFramebuffers(1, &fbo_accum);
2 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo_accum);
3 glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
4     GL_TEXTURE_2D, fbo_accum_texture, 0);
```

To produce a fade affect similar to the accumulation buffer approach two frame buffer objects had to be created, referred to as FBO1 and FBO2. It should be noted that blending has to be off for all of this. First, FBO1 is bound. Then, the texture from FBO2 is drawn on to FBO1 using a simple shader that uses a fade multiplier in the fragment stage. Next, the new points are drawn on FBO1 using a simple shader. After this, FBO2 is switched to the draw frame buffer and FBO1 is set as the read frame buffer. FBO1 is then copied to FBO2 using the frame buffer blit command. Finally, the default frame buffer is set as the draw frame buffer and FBO1 is copied to it. FBO1 is used for off screen rendering and FBO2 is used to hold a copy of the frame buffer between frame updates. Initially I tried to use the back buffer between frames but it is not in a floating point format and as a result the fade affect did not work this way.

Results

The results of this work focus heavily on the performance testing of different parts of the program. Bottle necks in performance were identified and remedied. File loading, compiler optimization, and various aspects of rendering performance were explored. Additionally, the potential for numerical error and how the solution was checked to make sure the error was low is also discussed. Finally what the final visualization program is capable of is shown last.

6.1 Initial Solution Testing

In order to verify the math code, a program was written to visualize a well known data set. This well known data set is referred to as the dipol data. Testing of the dipol data was with a very early version of the visualization software before it used shaders or frame buffer objects. For a validity check, the output of this program was compared to streamlines traced in Paraview for the dipol data set. In Figure 6.1 the dipol data is shown with stream lines traced in Paraview.

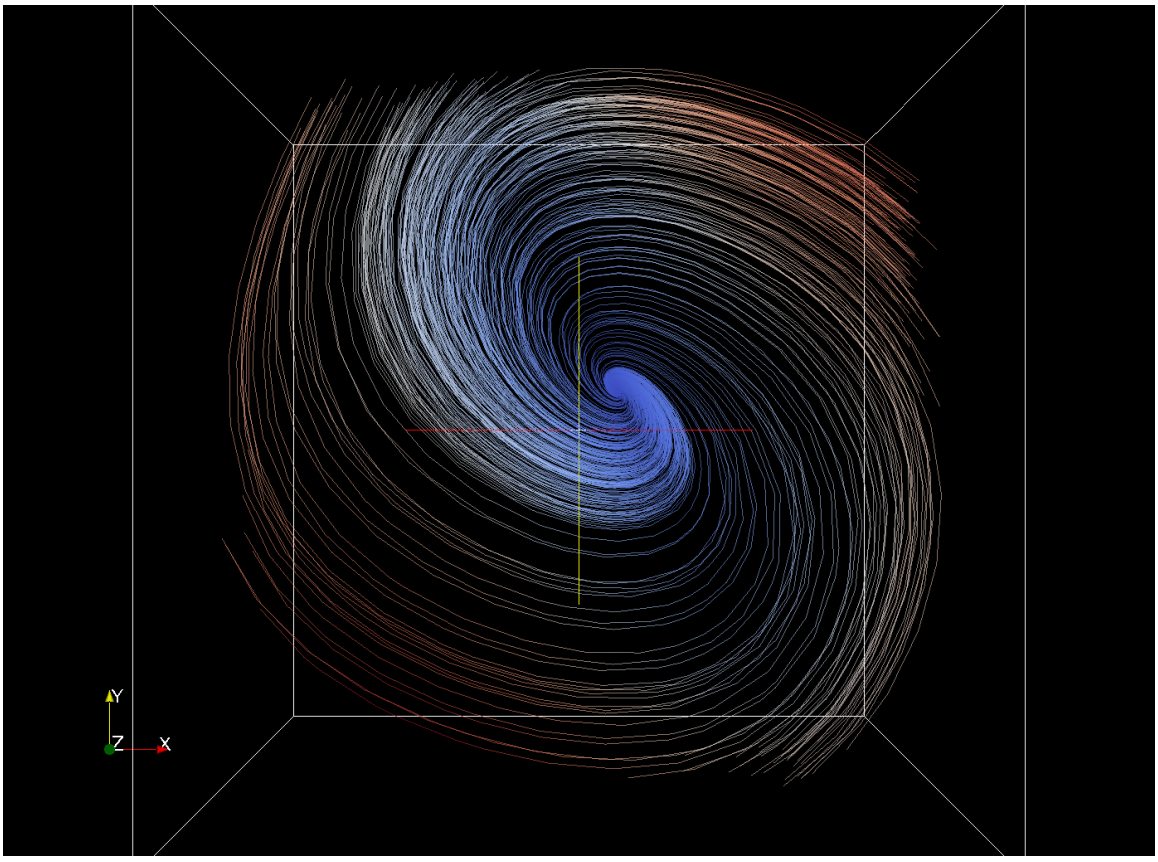

Figure 6.1: Dipol Stream Lines Traced in Paraview

Next, in Figures 6.2, 6.3 the initial stream line visualization is shown. The red lines in Figures 6.2 and 6.3 are a hedge hog plot of the velocity at the data points. These plots match well to the Paraview stream lines. At this point, a good question would be why not just use

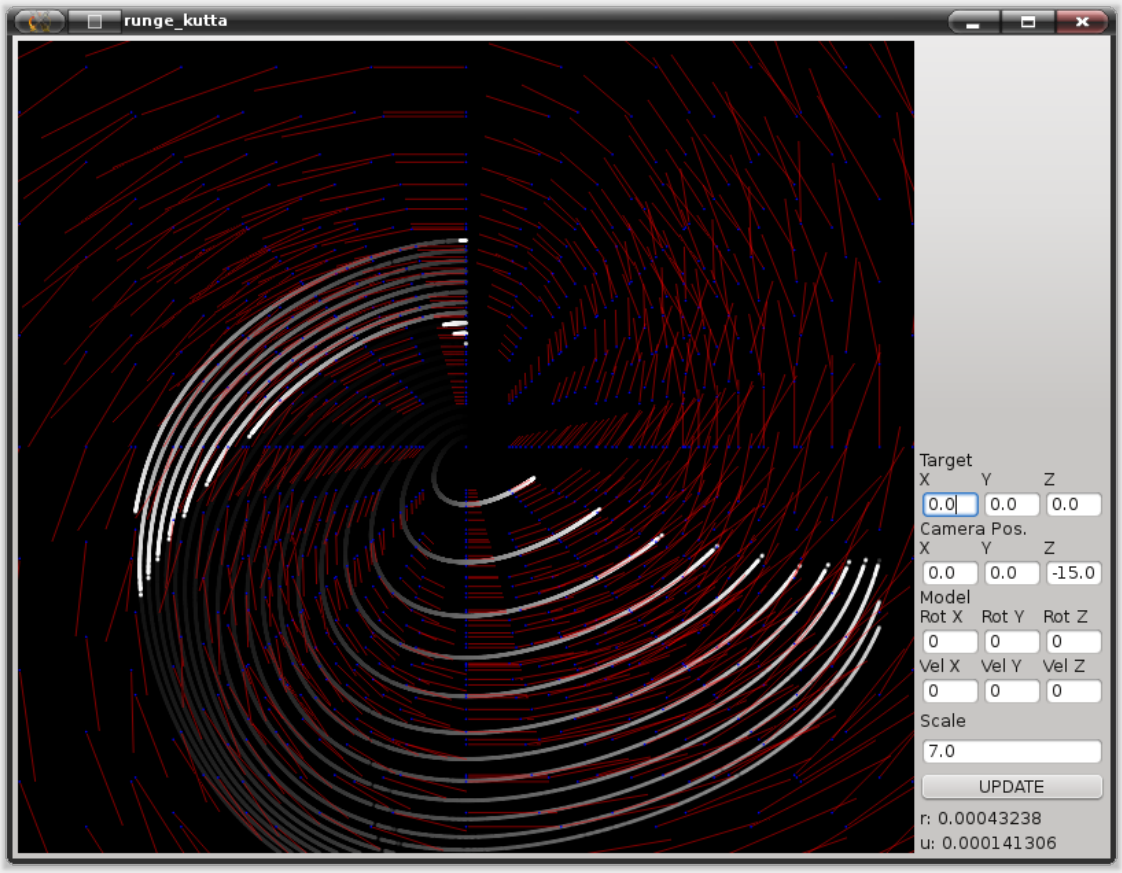

Figure 6.2: Dipol Stream Lines Traced in My Solution Part 1

Paraview to visualize the dragon fly flow data? The reason Paraview is not used is the dragon fly data is considerably more complex and a smoke trail effect with streamsurfaces was desired. Figure 6.4 shows how complex the dragon fly data is by tracing streamlines through one time interval of data.

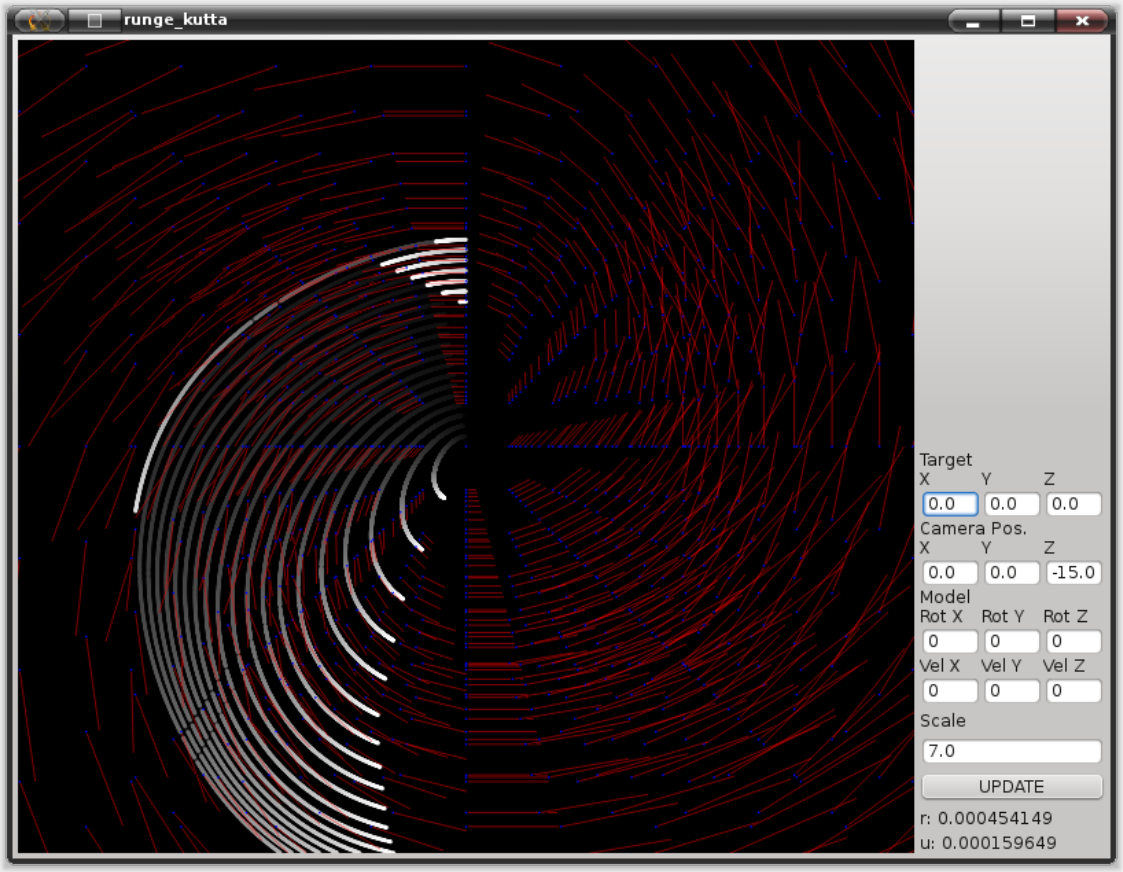

Figure 6.3: Dipol Stream Lines Traced in My Solution Part 2

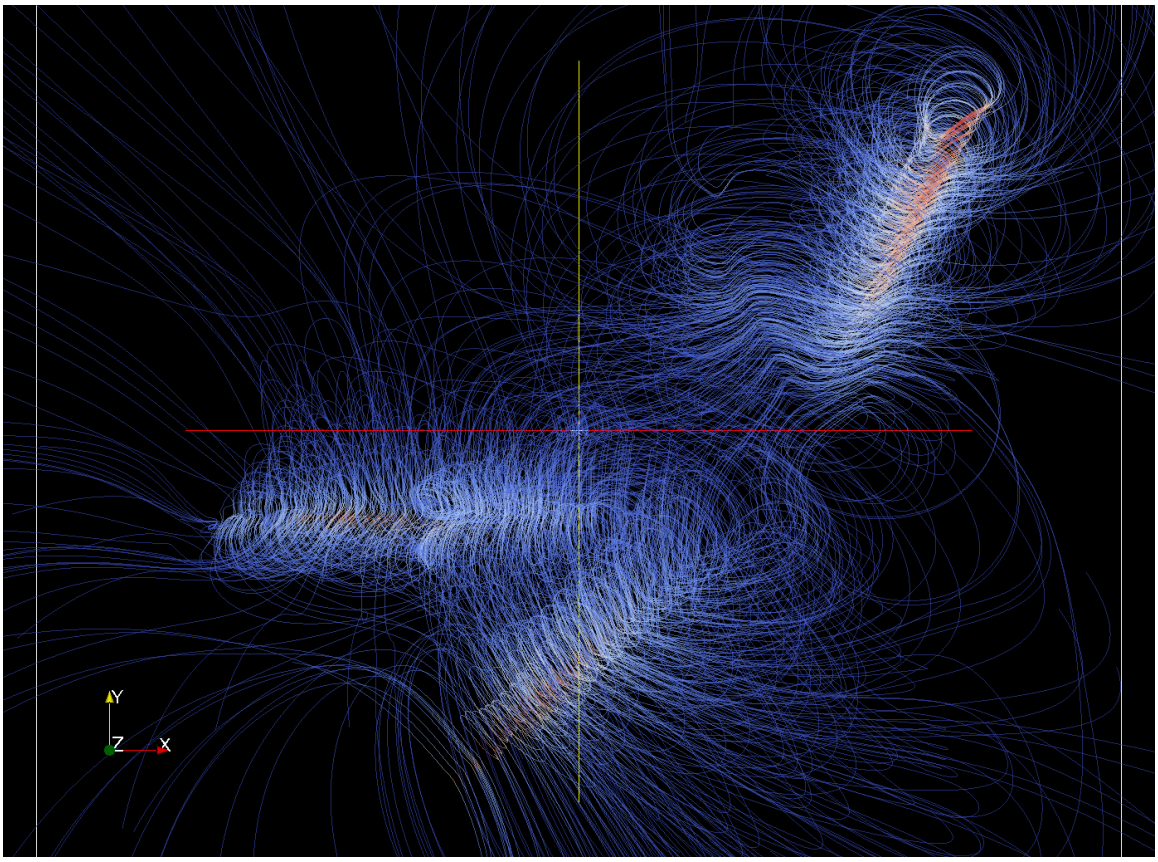

Figure 6.4: Dragon Fly Stream Lines Traced in Paraview

6.2 File Loading Comparison

Since one of the main functions of this project is loading large data files it was decided that testing the different methods of loading files on both Windows and Linux was important. For file loading there are the traditional fread() from the C standard library and ifstream::read() from the C++ standard library. Also, memory mapped file reads were tested. Memory mapped reads allow access to a file offset as if it is directly in memory. A program was written to test these three reading methods with the VTK binary data files.

For the test program, all memory that is needed is allocated first before the timing of the test starts. Then, the first file is parsed to find the offset of the binary data. Each method then uses this offset to read a block of binary data, in particular, the velocity data that is needed for the flow visualization. For this test, the size of the block of binary data is approximately 86 MB. The test was run for five different files, and run ten times per operating system. In an attempt to be fair, the native file system for each operating system was used, that is EXT4 for Linux and NTFS for Windows. The first machine tested, named F1, has two Western Digital Black drives in RAID0. Also, no antivirus was running for Windows during the tests. Here are the average results from the test in seconds.

Operating System	MMAP	ifstream::read()	fread()
openSUSE 12.2	0.0542191358	0.0323381409	0.0328242823
Windows 7 x64	0.0614432107	0.1702454273	0.0691770039

Table 6.1: File loading test on F1, Averages

The standard deviation for this test is shown next.

Operating System	MMAP	ifstream::read()	fread()
openSUSE 12.2	0.0011042724	0.0003225121	0.0008060155
Windows 7 x64	0.0014138286	0.0014979809	0.0013814431

Table 6.2: File loading test on F1, Standard Deviation

The next test was also carried out an on laptop named 535U3C with a much slower

CPU and hard drive. The hard drive for this test was a Hitachi 7200 RPM 500GB slim 7mm drive. Average loading times are as follows.

Operating System	MMAP	ifstream::read()	fread()
openSUSE 12.2	0.097619019	0.0620192797	0.0622527787
Windows 7 x64	0.1027432346	0.2537859979	0.1054068933

Table 6.3: File loading test on 535U3C, Averages

The standard deviation for this test is shown next.

Operating System	MMAP	ifstream::read()	fread()
openSUSE 12.2	0.0004983234	0.0007728442	0.0007970776
Windows 7 x64	0.0060013703	0.0018439132	0.0027589379

Table 6.4: File loading test on 535U3C, Standard Deviation

As can be concluded from the tables above, Linux is able to achieve faster load times in all cases. In Linux, memory mapped reads are about twice as slow as ifstream::read() and fread(). ifstream::read() and fread() perform about the same in Linux. In Windows, memory mapped reads are slightly faster than fread(), and ifstream::read() is more than twice as slow. Linux is about twice as fast as Windows at reading the block of binary data from the files.

6.3 Numerical Error

In any numerical integration method error is a primary concern. As previously stated, the 4th order Runge Kutta method should be the fastest but have less accuracy. Conversely, the 5th order methods should be slower but more accurate. Therefore, some tests were ran to see how the accuracy of these different methods compared.

For the accuracy testing a separate program was written. In order to make the visualization program modular, the code for integration in the vector field data was designed to be as separate as possible. This modularity lent itself well to breaking out the integration code for error and performance testing. Through this approach, the exact same code is tested providing accurate results. The error test program loads a single time interval of the dragon fly flow data. One of the streamsurface start position files (seedline files) is also specified. Once these files are loaded 10000 points are created along the seedline. For this testing the time step is fixed. Each of the three integration methods, RK4, RK5 with Cash Karp coefficients, and RK5 with Fehlberg were tested. During this testing the error was recorded at each iteration for each of the three methods and maximum value for over all of the points was taken. The following figure 6.5 shows a plot of the error for these methods.

It was surprising to find the error is very high for the first several iterations but drops off exponentially for all methods. As expected, the 5th order approaches converge much faster. Also, the Cash Karp coefficients converge at about the same rate at the Fehlberg coefficients but the final error is about an order of magnitude less for the Cash Karp method. For this testing the time step was set to 1.0 ms which is similar to the performance of the visualization program on an AMD FX-8150 CPU and Radeon 7950 graphics card. The high initial error in this test does explain a problem that was sometimes observed with the visualization program on slower computers, points would diverge at the start of the visualization in one particular area.

In order to investigate the error further the spatial gradient and the first and second

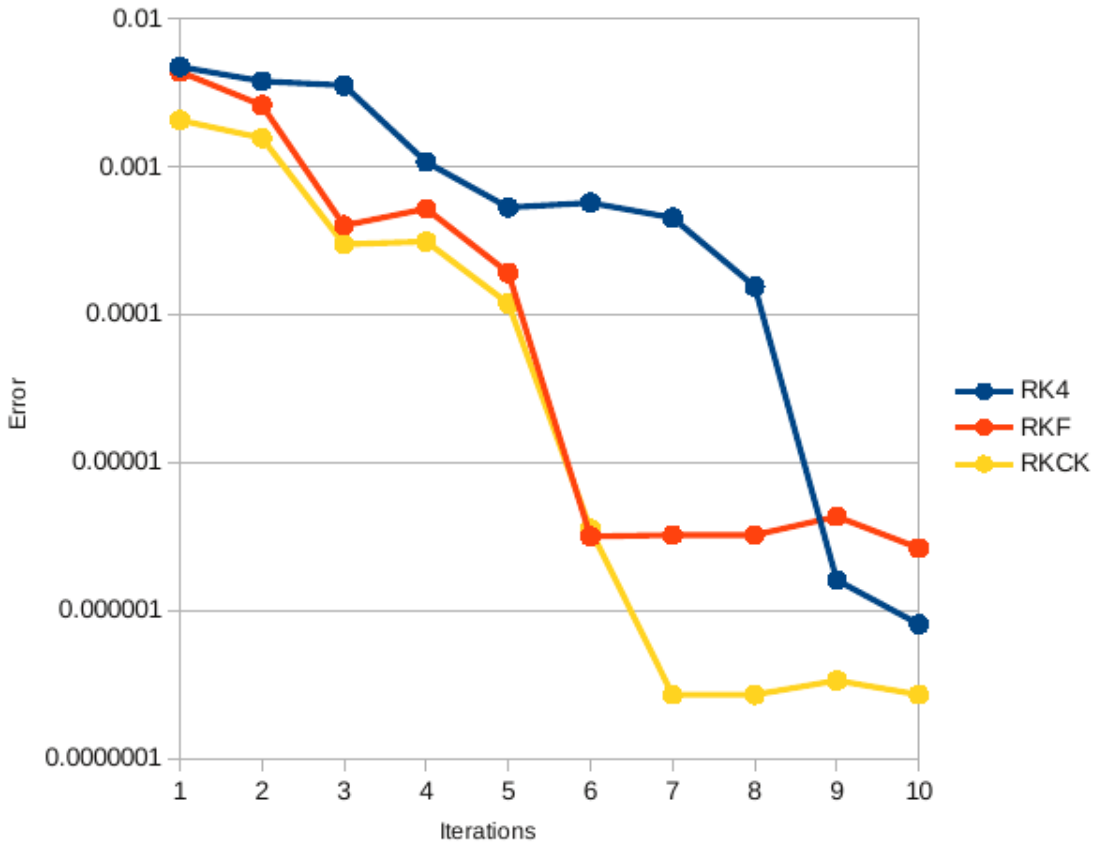

Figure 6.5: Comparison of Error in Runge Kutta Integration Methods.

gradients of the velocity were calculated. Through this testing some discontinuities near the area of higher error was discovered that could explain this high initial error and divergence problem. Figure 6.6 shows the error, position and velocity gradients along the line that the stream surface starts on.

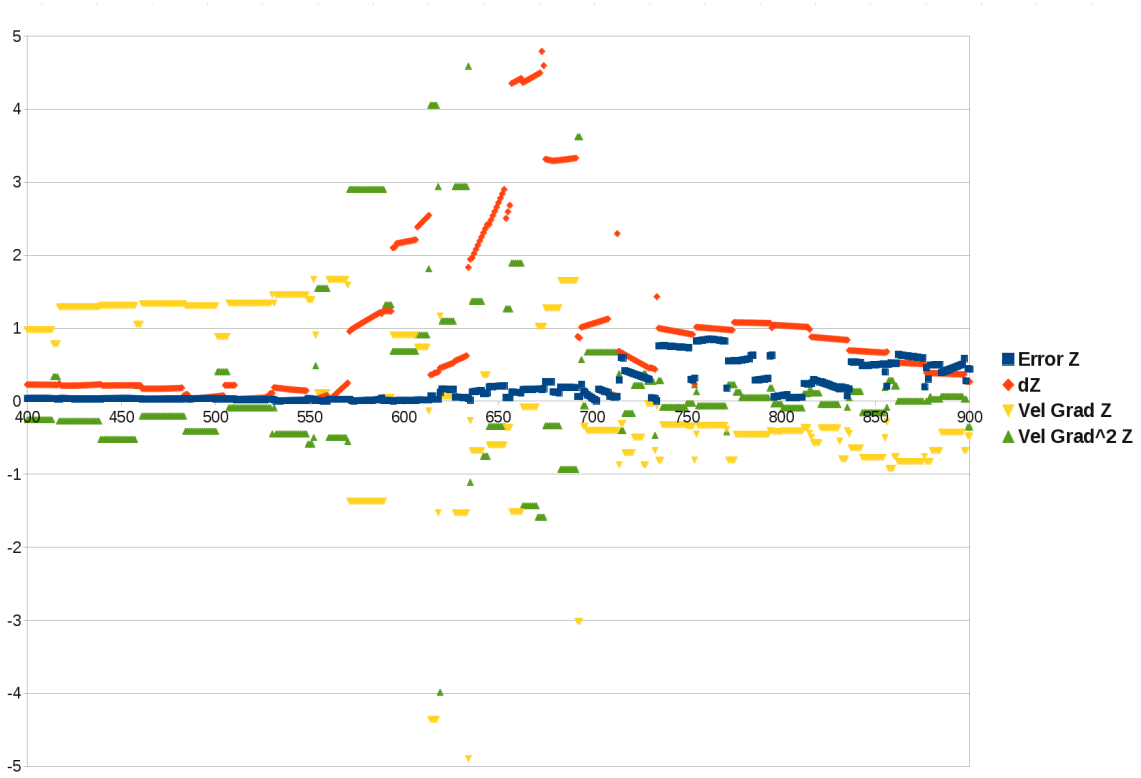

Figure 6.6: Comparison of Error, Position and Velocity Gradients

In Figure 6.6, Error Z is the error in the Z dimension. dZ is the spatial gradient in the Z dimension. Vel Grad Z is the velocity gradient in the Z dimension and Vel Grad 2 Z is the second velocity gradient in the Z dimension. Also, the error is scaled by 200 and the spatial gradient is scaled by 50. This scaling was purely done to make these items fit on the plot. The divergence in the integration happens around the 650 X value (these X axis values are points seeded for the calculations) in Figure 6.6. Given the rapidly changing velocity gradient in this region it is no wonder the error is higher near this area. This is simply a problem with the data and can not be overcome.

6.4 Compiler Optimization Testing

Different compiler optimizations were investigated for Windows and Linux to see which options provided the best results. This testing was done with a modified version of the math error test program. A vector math library, GLM, was used extensively in the math code. This library makes heavy use of intrinsics for vectorized floating point operations. The hope was that by using GLM as much as possible for the math code the various compilers would automatically use vectorized floating point instructions. This test was run on hardware supporting AVX floating point instructions, the most current for x86 hardware at the time of writing.

The test runs through the integration code for the fourth order Runge Kutta solution and the fifth order methods I tested with Felhberg and Cash Carp coefficients. 10000 points were initialized on a seed line and the next positions were calculated for 1000 iterations with a time step of 0.00001 seconds. Just one of the VTK data files was tested in this performance test. Five runs at each optimization were tested and the averages are shown for each hardware and optimization tested.

The first hardware test is a computer named F1 with an AMD FX-8150 CPU, 16GB DDR3 1866 MHz RAM, Western Digital Black hard drives in RAID 0, AMD Radeon 7950 graphics card with a 900 MHz core clock and 1250 MHz memory clock. Both operating systems are 64 bit and all code was compiled as 64bit. openSUSE uses GCC version 4.7.1 and kernel 3.4.11. Windows 7 Ultimate x64 with service pack 1 was used for windows testing and all updates were applied at the time of writing. Visual Studio 2010 SP1 and MinGW64 with GCC 4.7.1 were the compilers used for Windows. The CPU governor was set to performance for Linux and the power saving mode was set to High Performance for Windows. Consequently, the CPU runs at 3.9 GHz with these performance modes set. HPC (High Performance Computing) mode was also enabled in the BIOS. This HPC mode sets the top three CPU P states to the highest clock frequency. The results for computer F1 are

RK4 Time	RKF Time	RKCK Time	Operating System and Optimizations
2.088422	3.377116	3.441972	Linux -O2 -ffast-math -march=native amdlibm
2.092506	3.386268	3.444376	Linux -O2 -ffast-math -march=native
2.104396	3.417744	3.405242	Linux -O2 -march=native amdlibm
2.112878	3.407816	3.400398	Linux -O2 -march=native
2.128004	3.421732	3.404982	Linux -O3 -march=native
2.150666	3.421898	3.370406	Linux -O2
2.155638	3.446926	3.429318	Linux -O2 -ffast-math -march=native amdlibm *
2.18367	3.545358	3.581074	Linux -Ofast -march=native
2.2294	3.503782	3.478262	Linux -O2 -march=barcelona
2.236514	3.539422	3.512234	Linux -O2 -march=athlon64
2.36889	3.712304	3.65373	mingw64 -O2 -march=bdver1
2.369368	3.702368	3.647548	mingw64 -O3 -march=bdver1
2.452192	3.866622	3.92477	mingw64 -O2 -march=bdver1 -ffastmath
2.46743	3.934298	3.86532	mingw64 -O3
2.542574	4.052832	3.972278	mingw64 -O2
2.58107	4.050654	4.033692	VS2010 /O2 /Oi /GL /favor:AMD /arch:AVX
2.584042	4.030874	4.008586	VS2010 /O2 /Oi /GL /favor:AMD /arch:AVX amdlibm
2.585166	4.051624	4.022596	VS2010 /O2 /Oi /GL /arch:AVX
2.61549	4.15656	4.111576	VS2010 /O2 /Oi /GL /arch:AVX /fp:fast
2.636216	4.139902	4.213498	mingw64 -O3 -march=bdver1 -ffastmath
2.978906	4.55239	4.530582	VS2010 /O2 /Oi /GL /favor:INTEL
2.989004	4.535984	4.479508	VS2010 /O2 /Oi /GL
2.989074	4.55099	4.51706	VS2010 /O2 /GL
2.99463	4.57004	4.539286	VS2010 /O2 /Oi /GL /favor:AMD
6.091216	9.885638	9.776148	Linux -Os -march=native
23.16124	37.11162	36.93814	Linux -O0 -march=native

Table 6.5: Execution Time for Various Compiler Optimizations on F1

shown in Table 6.5.

The results in Table 6.5 are ordered by lowest running time for the 4th Order Runge Kutta (RK4) solution. Linux performs much faster than Microsoft Windows in this test. The -ffast-math option provides the best results when coupled with other options, but this is not reasonable to use for scientific computing due to the potential error in the calculations. However, -ffast-math is relevant for computer games or simulations where exact accuracy is not a requirement. AMD's libm was used and provided the best results in Linux, but this library did not perform as well in Windows. Also of interest, the -O2 optimization performed slightly better in Linux than the -O3 optimization, this was unexpected. The

'amdlibm *' test case uses AMD's suggested compiler flags [29], [30] and did quite poorly. Linux without optimizations (-O0) ran more than ten times slower than the other options. Linux with just the -O2 optimization worked quite well, and this code would run on any CPU. On the Windows side forcing AVX optimizations helped quite a bit for both Visual Studio and MinGW64. MinGW64 was considerably more fast than Visual Studio, with just default optimizations and with AVX specific optimizations. The fast math option in Windows did not provide a performance boost.

The next system tested is a Samsung model 535U3C laptop. This system features the new AMD A6-4455M CPU with a Trinity core. This CPU is a low power CPU close to a netbook CPU, it is clocked at 2.1 GHz and supports AVX instructions. This system has 8 GB of DDR3-1333 RAM also. The results for the 535U3C are shown in Table 6.6.

As expected, most likely due to the low CPU frequency, this system was the slowest. The results were very similar to the computer named F1 with the FX-8150. Both of these computers use the AMD Bulldozer core, just different revisions, so it stands to reason that they would be similar.

The next system tested is an HP DV7 laptop with an AMD A8-3500M clocked at 1.5 GHz (2.4 GHz boost) and 16GB DDR3-1333 RAM. This CPU does not support AVX instructions, but it is not too old either so performance should be reasonable. On the other hand, the clock frequency is rather low, so it was interesting to test. For the sake of time MinGW64 tests were omitted. This performance results for this system are shown in Table 6.7.

This system performed well with -Ofast and -O2. Unfortunately, the AMD libm library did not help the performance much. Forcing the the optimizations to the athlon64 or barcelona instruction sets hurt performance a lot in Linux. For Windows, the speed was quite a bit behind Linux again. The /favor:AMD or /favor:INTEL options achieved better results than anything else in Windows. All things considered, the DV7 with it's AMD

RK4 Time	RKF Time	RKCK Time	Operating System and Optimizations
4.082102	6.683542	6.735498	OpenSUSE -O2 -ffast-math -march=native amdlibm
4.115224	6.722446	6.808172	OpenSUSE -O2 -ffast-math -march=native
4.16834	6.70011	6.681242	OpenSUSE -O2 -march=native amdlibm
4.209292	6.743984	6.690026	OpenSUSE -O2 -march=native
4.22444	6.72499	6.708882	OpenSUSE -O3 -march=native
4.293362	6.801748	6.784528	OpenSUSE -O2 -ffast-math -march=native amdlibm *
4.301464	6.860386	6.787604	OpenSUSE -O2
4.328942	6.845868	6.780406	OpenSUSE -O2 -march=barcelona
4.379118	6.87754	6.810116	OpenSUSE -O2 -march=athlon64
4.383482	6.941996	7.011702	OpenSUSE -Ofast -march=native
4.456936	6.95261	6.969896	mingw64 -O2 -march=bdver1
4.53838	7.028118	6.836454	mingw64 -O3 -march=bdver1
4.5819	7.245358	7.501414	mingw64 -O2 -march=bdver1 -ffast-math
4.748842	7.525232	7.398626	mingw64 -O2
4.769032	7.453304	7.400686	Windows /O2 /Oi /GL /favor:AMD /arch:AVX
4.773472	7.464862	7.41783	Windows /O2 /Oi /GL /favor:AMD /arch:AVX amdlibm
4.777946	7.45338	7.418516	Windows /O2 /Oi /GL /arch:AVX
4.789242	7.453634	7.42717	Windows /O2 /Oi /GL /arch:AVX /fp:fast
4.832808	7.721822	7.404186	mingw64 -O3
5.092128	8.04299	8.077462	mingw64 -O3 -march=bdver1 -ffast-math
5.414596	8.398704	8.31596	Windows /O2 /Oi /GL /favor:INTEL
5.416922	8.382854	8.3331	Windows /O2 /Oi /GL
5.430736	8.37674	8.332614	Windows /O2 /GL
5.451602	8.386788	8.337312	Windows /O2 /Oi /GL /favor:AMD
11.6315	18.68696	18.57274	OpenSUSE -Os -march=native
38.56376	60.3957	60.80114	OpenSUSE -O0 -march=native

Table 6.6: Execution Time for Various Compiler Optimizations on 535U3C

A8-3500M CPU did not perform too poorly compared to the other systems tested.

The last machine tested is an older laptop referred to as G51. This laptop has an Intel Core 2 Duo T9600 CPU clocked at 2.8 GHz, NVidia GeForce GTX 260M graphics card and 4GB of RAM. The results of the compiler optimization performance test is shown in Table 6.8.

On this older Intel CPU Linux still out-performed Windows, but not by as large of a margin. Oddly enough, the -Ofast option performed the best here, while on the previous AMD CPUs this optimization did quite poorly. As with other systems, the plain -O2 Linux

RK4 Time	RKF Time	RKCK Time	Operating System and Optimizations
3.2876	5.286336	5.332136	OpenSUSE -Ofast -march=native
3.35752	5.359754	5.304484	OpenSUSE -O2 -march=native
3.383986	5.300908	5.426126	OpenSUSE -O2 -ffast-math -march=native
3.437196	5.265486	5.23788	OpenSUSE -O2
3.434844	5.50516	5.409162	OpenSUSE -O2 -ffast-math -march=native amdlibm *
3.52398	5.554268	5.365346	OpenSUSE -O2 -march=native amdlibm
3.5285	5.478982	5.371472	OpenSUSE -O2 -ffast-math -march=native amdlibm
3.58822	5.420892	5.458982	OpenSUSE -O3 -march=native
3.80055	5.737886	5.785332	OpenSUSE -O2 -march=athlon64
3.88524	5.551818	5.55584	OpenSUSE -O2 -march=barcelona
4.917928	7.77161	7.759908	Windows /O2 /Oi /GL /favor:AMD
4.93469	7.78923	7.74307	Windows /O2 /Oi /GL /favor:INTEL
4.967082	7.73893	7.548652	Windows /O2 /GL
5.095038	8.059162	7.800146	Windows /O2 /Oi /GL
9.874134	16.46528	15.56924	OpenSUSE -Os -march=native
44.87606	70.35258	70.86104	OpenSUSE -O0 -march=native

Table 6.7: Execution Time for Various Compiler Optimizations on DV7

RK4 Time	RKF Time	RKCK Time	Operating System and Optimizations
2.594882	4.099574	4.056218	OpenSUSE -Ofast -march=native
2.598598	4.095468	4.062234	OpenSUSE -O2 -ffast-math -march=native amdlibm
2.599154	4.094114	4.060358	OpenSUSE -O2 -ffast-math -march=native
2.642978	4.145124	4.138642	OpenSUSE -O3 -march=native
2.67363	4.176926	4.137908	OpenSUSE -O2 -ffast-math -march=native amdlibm *
2.715116	4.232114	4.239952	OpenSUSE -O2 -march=native amdlibm
2.71807	4.231684	4.251706	OpenSUSE -O2 -march=native
2.759052	4.38796	4.318838	OpenSUSE -O2 -march=barcelona
2.777986	4.402238	4.352598	OpenSUSE -O2 -march=athlon64
2.840254	4.459768	4.316948	OpenSUSE -O2
3.186666	5.000578	4.96198	Windows /O2 /Oi /GL /favor:INTEL
3.188506	5.002366	4.959632	Windows /O2 /Oi /GL
3.190892	4.993296	4.959072	Windows /O2 /GL
3.22732	4.991098	4.952876	Windows /O2 /Oi /GL /favor:AMD
6.908324	11.0482	10.95846	OpenSUSE -Os -march=native
35.00178	56.28832	56.25486	OpenSUSE -O0 -march=native

Table 6.8: Execution Time for Various Compiler Optimizations on G51

test did not perform well either. Surprisingly, AMD's libm implementation did very well even though this is an older Intel CPU. On the Windows side anything but the /favor:AMD option did ok. Anything with /arch:AVX failed to run for this CPU since it does not support

AVX instructions.

In general, Linux performs much better than Windows for this type of work load which is representative of Scientific computing. Linux usually performs best with -O2 optimizations and -march-native, with just -O2 close behind if the binaries need to run on different types of CPUs. For Windows using /favor:INTEL would not be a bad general use option. If AVX instructions are available for Windows, the /arch:AVX compiler flag should be used, assuming the binaries will not be run on non-AVX CPUs.

6.5 Data Container Performance

The visualization of a streamsurface requires keeping track of and updating a large number of points. These points need to be in order with respect to the streamsurface so it is easy to determine the space between them. Also, points need to be added when the gap is past a threshold and removed when the gap is under a threshold. All of this has to happen every time a frame is drawn. It was discovered using AMD's Code Analyst software, that a large amount of time was spent in the C++ list container that was being used to hold the points. To test impact of the C++ list container, the list container was temporarily replaced with a static array and the adding of points was stopped. As expected from the Code Analyst information, a big increase in performance happened when the C++ list container was removed.

So, in order to have something more useful than a static array and faster than a C++ list a custom data container was created that allocates a large memory pool when the program is started. Nodes in this pool contain the data for a point and some administration information indicating whether the node is in use and where the previous and next nodes are. This in affect is a doubly linked list with a much smaller insert and delete time. The container also keeps track of the next available node. This container was compared against the C++ list container on Linux and Windows, and the results are shown in Table 6.9 and Table 6.10.

Container	Allocate	Insert	Access	Delete	Free
list avg	2.28E-004	1.02E-005	1.09E-005	6.64E-006	8.08E-005
pool avg	9.74E-005	1.16E-006	1.05E-005	1.08E-006	1.90E-005
list std dev	1.01E-005	1.23E-006	1.32E-006	5.91E-007	9.50E-006
pool std dev	8.85E-006	1.44E-008	1.76E-007	7.10E-008	1.50E-006

Table 6.9: Data Container Performance on Linux with FX-8150 CPU

Pool represents the data from the pool list container and list is the stdc++ list container. This was tested on the computer referenced other places in this paper as F1 with an AMD FX-8150 CPU. The big interesting result here is the pool container was able to achieve a ten

Container	Allocate	Insert	Access	Delete	Free
list avg	2.95E-004	1.25E-005	1.91E-005	7.65E-006	2.48E-004
pool avg	8.88E-005	1.13E-006	1.50E-005	1.87E-006	5.10E-007
list std dev	8.03E-006	2.70E-006	9.61E-006	3.13E-006	7.98E-005
pool std dev	9.04E-007	0.00E+000	7.94E-006	8.90E-007	1.14E-007

Table 6.10: Data Container Performance on Windows with FX-8150 CPU

fold performance increase for inserting new objects into the container for both Windows and Linux. Removing objects is about six to seven times faster on Linux and about four times as fast on Windows. This Pool List container greatly helps the performance of the final program.

6.6 Rendering Comparison

A major component of this project is the performance of the FBO based fade affect. In the interest of a comparison the FBO method was compared to an accumulation buffer fade affect. For fairness, the same code is used to render both, including the same shaders to draw the points. A fixed number of 512 points are drawn to simulate rendering a streamsurface. The points are not kept in a line as is normal for a streamsurface in this test because the purpose is to purely test rendering performance. Also, the same code is used for both Windows and Linux. Windows 7 Ultimate 64bit was used for Windows testing and OpenSUSE 12.2 with Linux kernel 3.4.11 and KDE 4.9 were used for Linux. OpenGL compositing for KDE was disabled in Linux unless otherwise noted. The CPU frequency governor was set to performance in Linux. For Windows the power profile was set to High Performance. This test uses the Qt library, specifically Qt 4.8.4 at the time of testing. GCC 4.7.1 is used for Linux and Visual Studio 2010 SP1 is used for Windows. All builds are 64bit with O2 optimizations. AVX instructions are enabled when available on a CPU. For AMD graphics cards, the Catalyst 13.1 driver for both operating systems. For NVidia graphics cards, the most current driver for each operating system was used, specifically the 310.90 driver for Windows and the 313.18 driver for Linux. The following resolutions were tested where supported by the hardware: 640x480, 800x600, 1024x768, 1280x720, 1366x768, 1600x900, 1920x1080, and 2560x1440.

The first computer tested is referred to as F1 and has an AMD FX-8150 CPU, AMD Radeon HD 7950 graphics card, 16 GB RAM clocked at 1866 Mhz, and a two disk Raid 0 array comprised of Western Digital Black drives. This system supports a maximum resolution of 2560x1440. The following Figure 6.7 shows the performance of the different methods, resolutions, and operating systems.

As shown in Figure 6.7, the FBO fade affect works better than the accumulation buffer fade affect at resolutions higher than 1280x720 on this hardware for Linux and resolutions

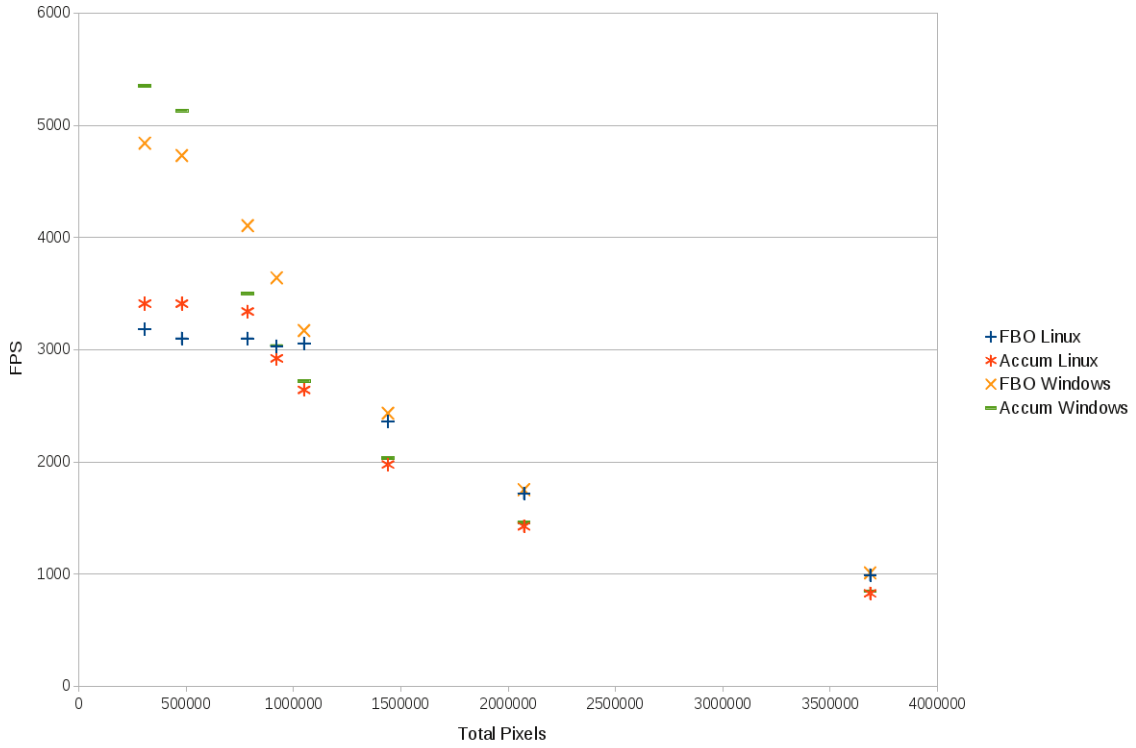

Figure 6.7: Rendering Performance on Computer F1

higher than 1024x768 for Windows. On this hardware Windows performed better than Linux at low resolutions. The performance was close at high resolutions but Windows still did better. Linux exhibited an odd flat line in performance for lower resolutions up to 1024x768.

For the next test an Asus G51 laptop was used. This machine has an Intel Core 2 Duo T9600 CPU, NVidia GTX 260M graphics card, 4 GB RAM, and a single Western Digital Black drive. This laptop supports a maximum resolution of 1920x1080. It should be noted that this machine is rather old, about 4 year old that is. The performance test results for rendering are shown next in Figure 6.8.

Here again the FBO fade affect performs better than the accumulation buffer fade affect. Linux did not exhibit poor performance at lower resolutions on this hardware either. In fact, Linux performed better all around than Windows. The performance was fairly

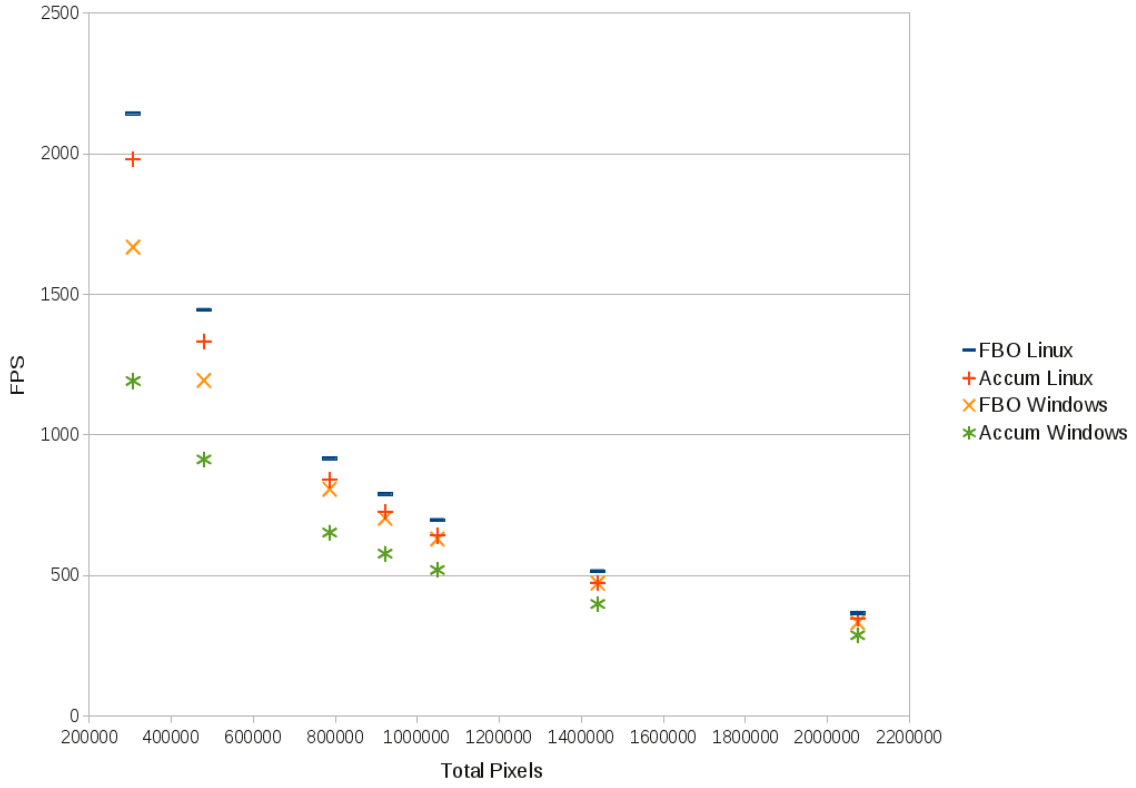

Figure 6.8: Rendering Performance on Computer G51

close for all cases at the highest resolution. This may be due to the slower DDR3 RAM in this graphics card. It would be reasonable to suspect AMD's driver is not quite up to the performance level in Linux than it is in Windows due to the results from the NVidia laptop testing and previous AMD system tests.

The last hardware tested is a Samsung 535U3C laptop. The 535U3C has an AMD A6-4455M CPU/APU with a Radeon HD 7500G graphics card on board. This graphics card shares system RAM which is 8GB of 1333 Mhz RAM. This laptop supports a maximum resolution of 1366x768. The rendering performance testing for this laptop is shown in Figure 6.9.

Here again the FBO fade affect performed better than the accumulation buffer fade affect. This laptop exhibited the same odd behavior in Linux where the performance is lower in lower resolutions than 1024x768 and then beats Windows when the resolution is

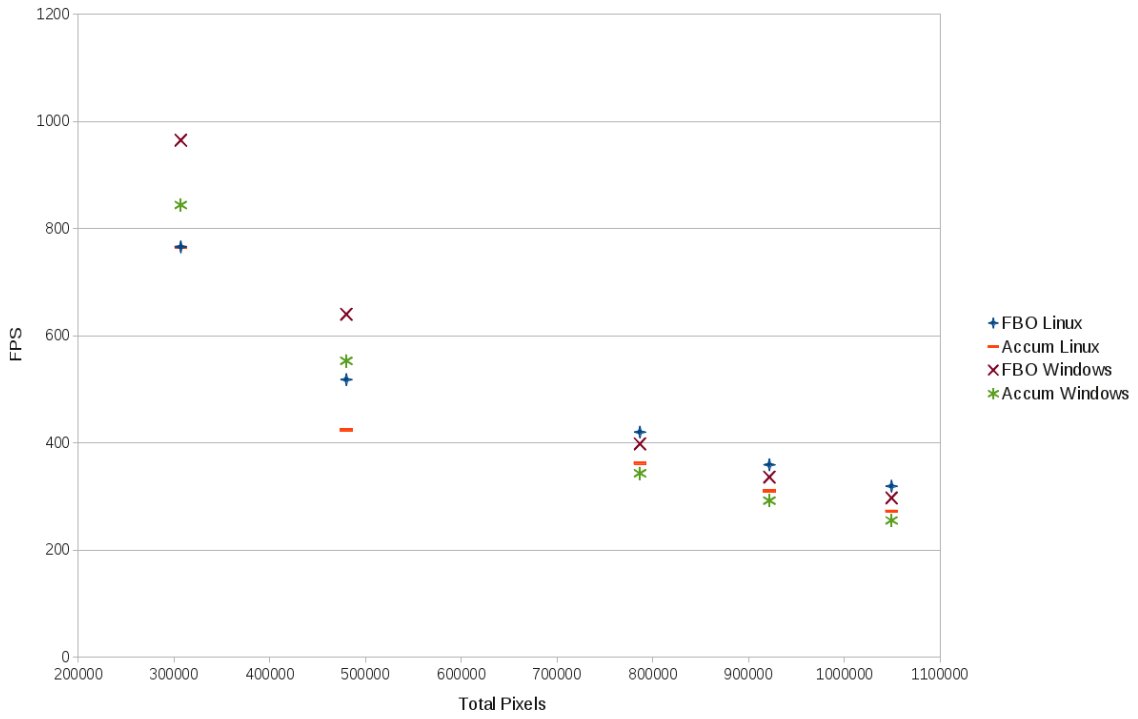

Figure 6.9: Rendering Performance on Computer 535U3C

1024x768 or higher. This probably would not matter for most scientific visualization software because the resolution should be much higher. This laptop, even though the hardware is not the greatest, was still able to achieve 320 FPS at it's native resolution, a testament to the high performance of this method.

In every case the FBO fade affect is faster than the accumulation buffer fade affect. This higher performance is one of the reasons to switch to the FBO fade affect, the other being the accumulation buffer is not supported in modern OpenGL specifications. Windows and Linux performance was very close with Windows performing slightly better for one graphics card that was tested and Linux performing slightly better for all of the other graphics cards that were tested.

6.7 Visualization Results

The streamsurface visualization program can render eight streamsurfaces at one time. This multi-threading is possible because threads are used for the calculations that update the point positions. These threads all have to join to the main thread before the new point positions are sent to the graphics card. Consequently, the program is sped up quite a bit since more time is spent updating the point positions rather than rendering. Figures 6.10, 6.11, 6.12, and 6.13 show multiple streamsurfaces being rendered for the dragonfly data set. The streamsurfaces start out with approximately 600 points and can increase to 2048 each. On an AMD FX-8150 CPU, AMD Radeon HD 7950 GPU, and OpenSUSE Linux the visualization program is able to achieve 245 FPS at a resolution of 2560x1440. One oddity that was noticed is the CPU load average stays at 400 percent, not 800 percent as expected. One possible explanation for this is the FX-8150 CPU only has 4 floating point units even though it has 8 integer units. Since the work load is almost entirely floating point this would account for the load average only being 400 percent.

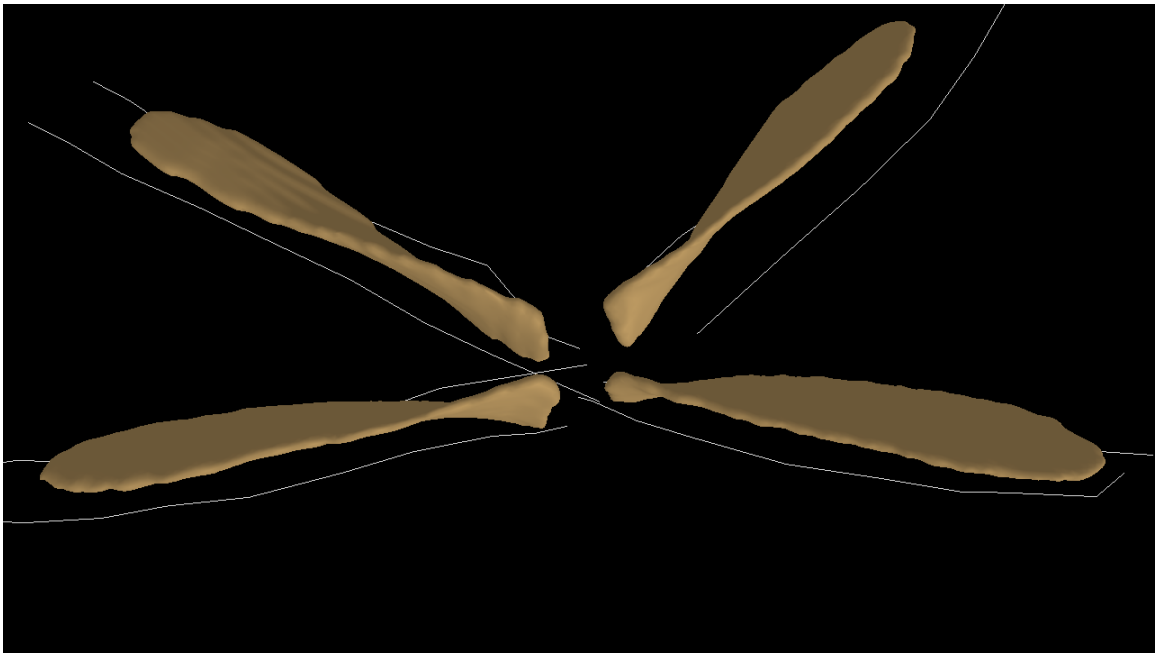

Figure 6.10: Eight Streamsurfaces 1

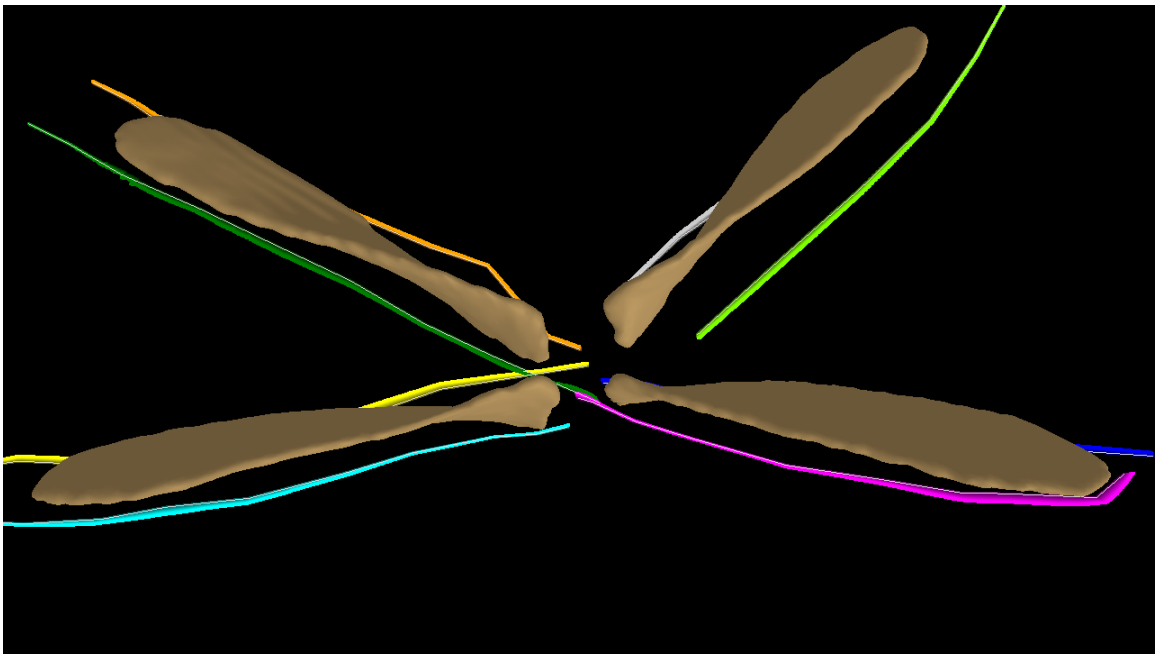

Figure 6.11: Eight Streamsurfaces 2

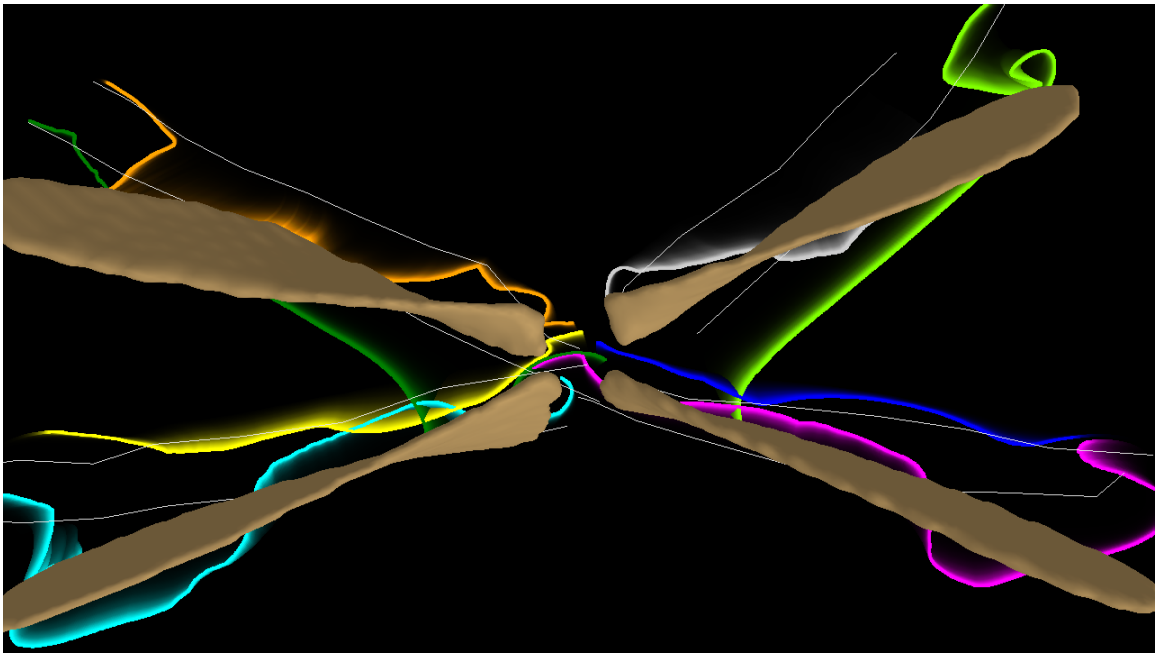

Figure 6.12: Eight Streamsurfaces 3

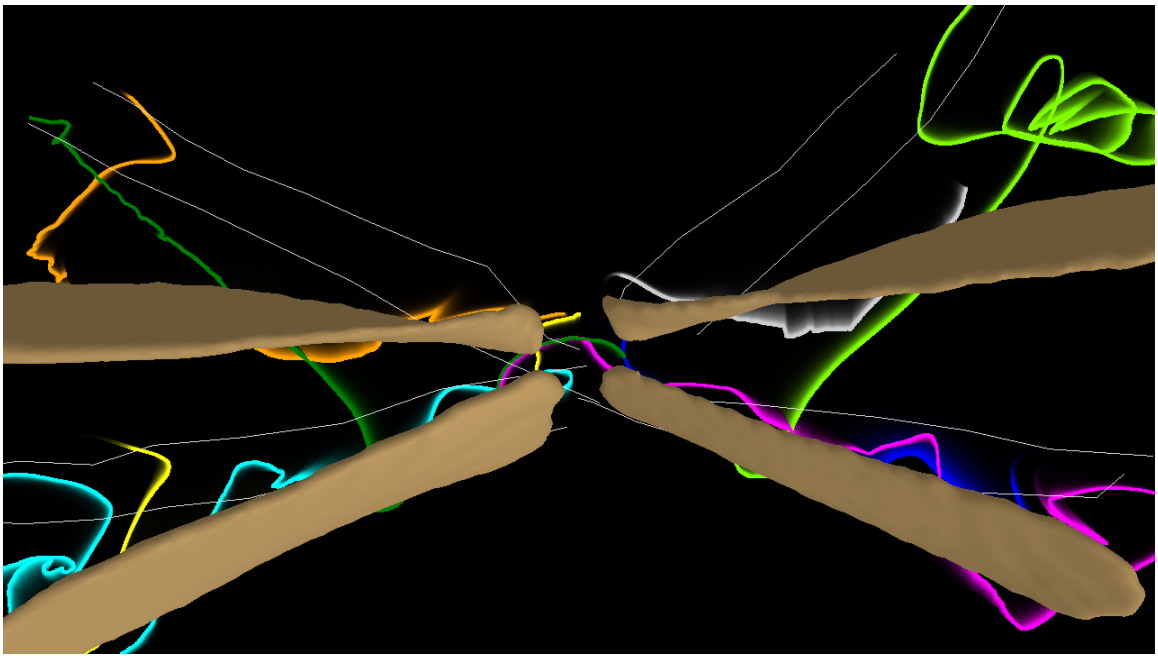

Figure 6.13: Eight Streamsurfaces 4

The next series of figures show how the streamsurface smoke affect can visualize vorticies. In Figures 6.14, 6.15, and 6.16 the streamsurface is just starting and is not distorted too much. Figures 6.17, 6.18, 6.19, 6.20, 6.21, 6.22 show a vortex starting to form in the lower left. Next, a vortex starts to form in the upper left shown in Figures 6.23, 6.24, 6.25. In Figures 6.26, 6.27, 6.28, and 6.29 the vortex continues to form. The point limit of 2048 has been reached and you can see the streamsurface break apart. This is a major limitation for this visualization approach.

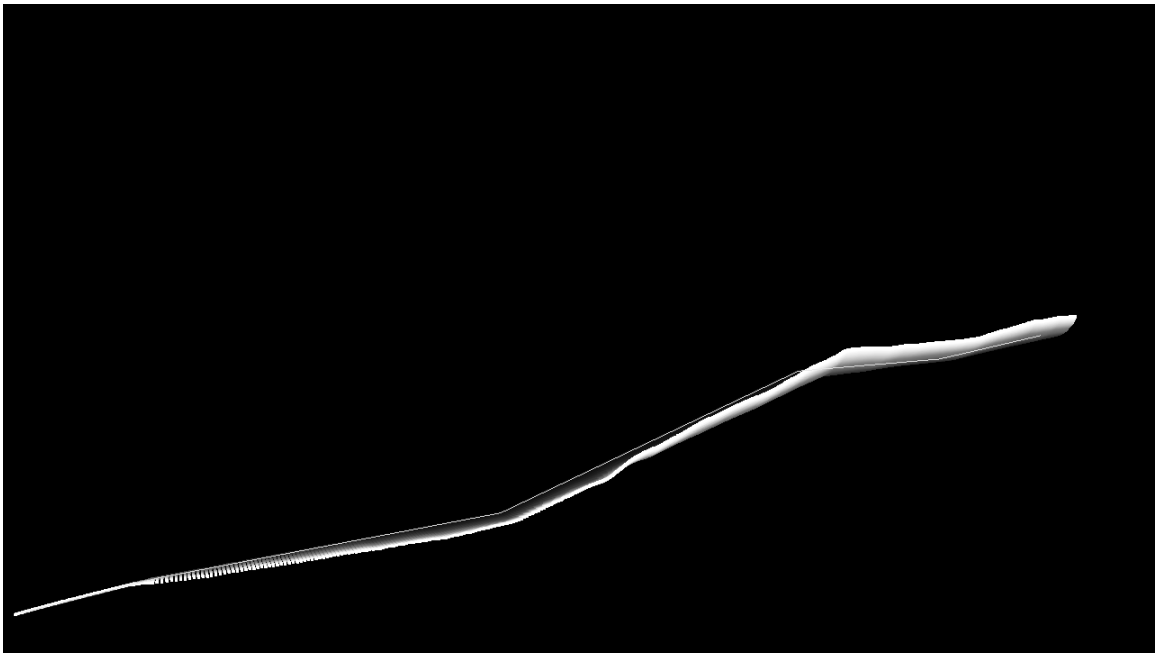

Figure 6.14: Streamsurface Vortex Visualization 1

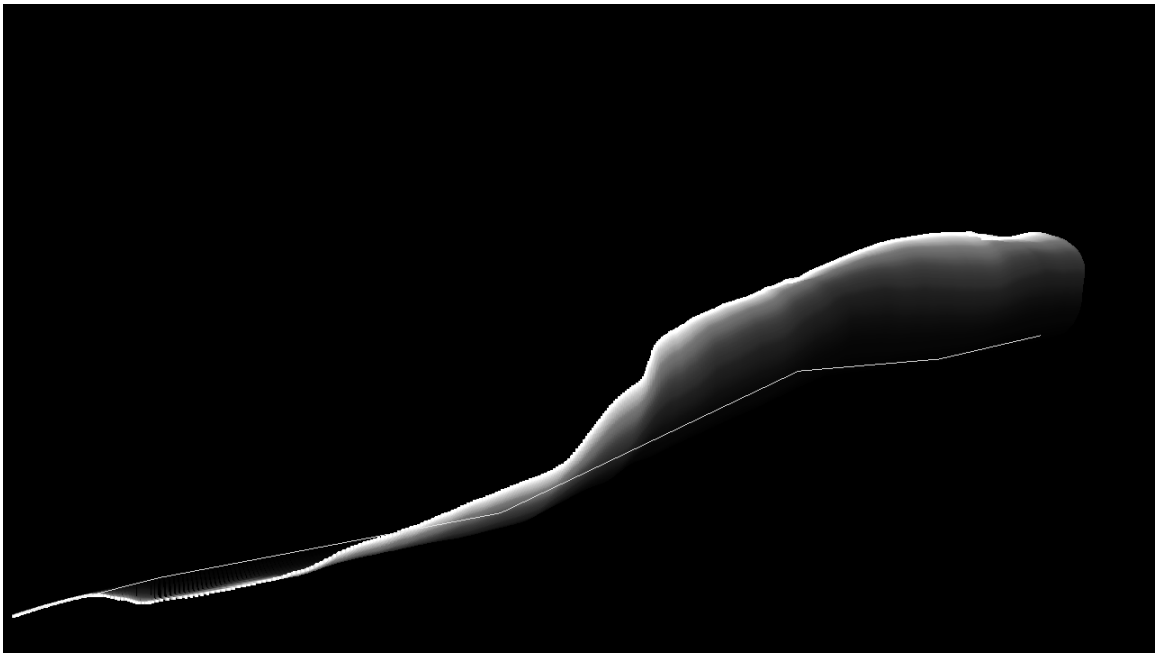

Figure 6.15: Streamsurface Vortex Visualization 2

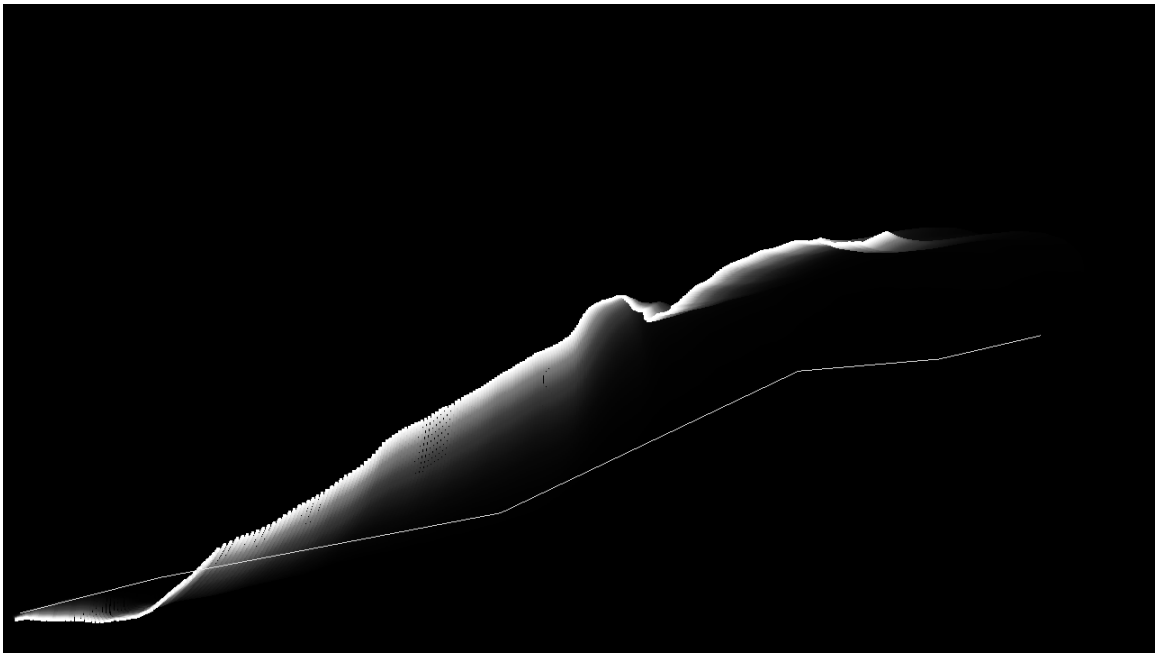

Figure 6.16: Streamsurface Vortex Visualization 3

Figure 6.17: Streamsurface Vortex Visualization 4

Figure 6.18: Streamsurface Vortex Visualization 5

Figure 6.19: Streamsurface Vortex Visualization 6

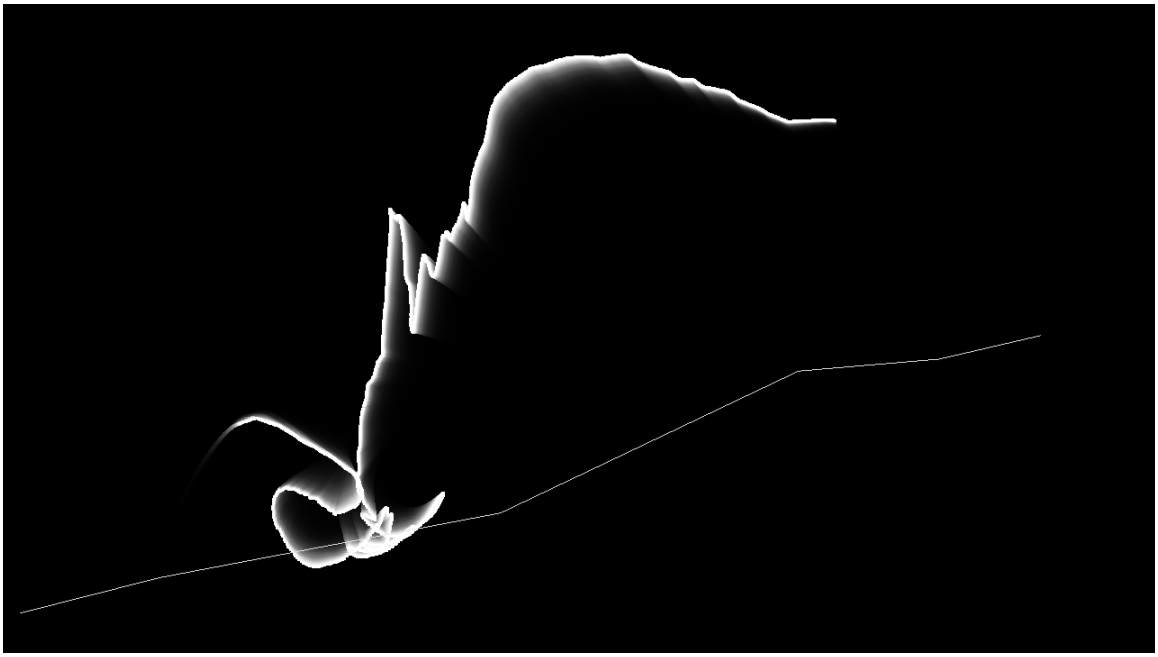

Figure 6.20: Streamsurface Vortex Visualization 7

Figure 6.21: Streamsurface Vortex Visualization 8

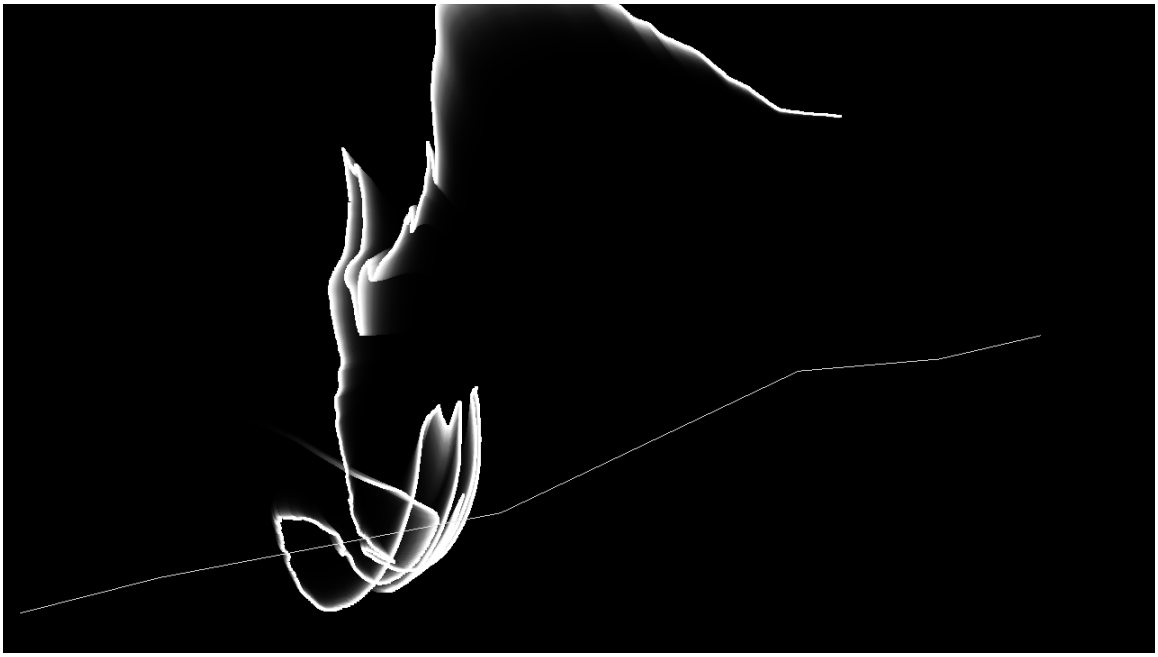

Figure 6.22: Streamsurface Vortex Visualization 9

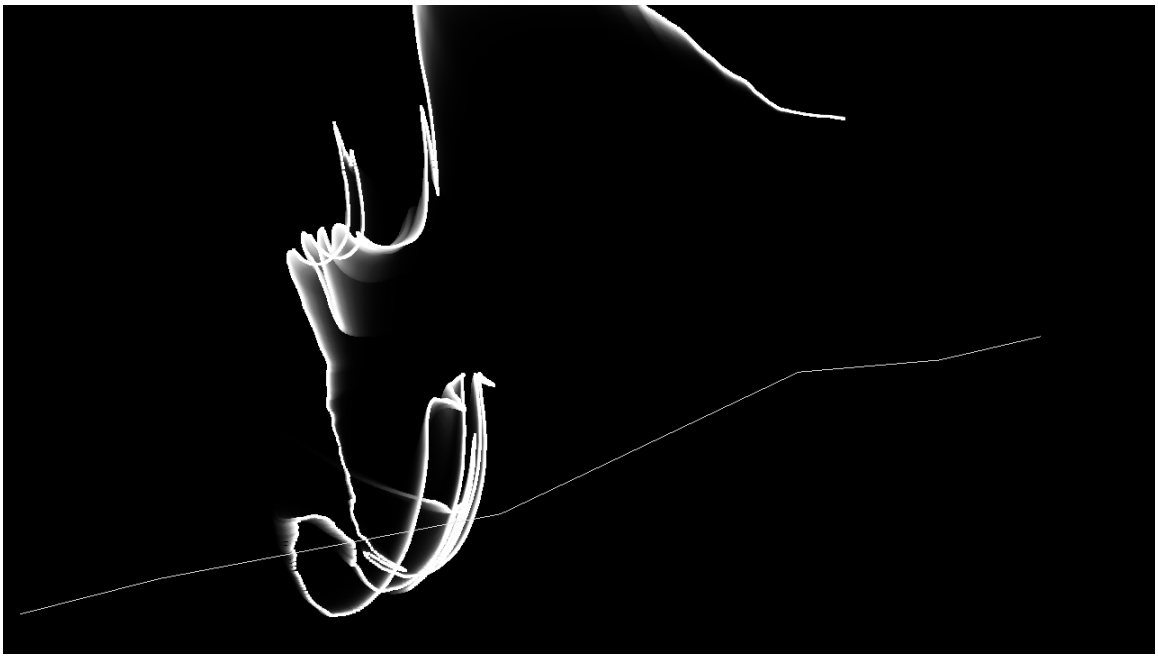

Figure 6.23: Streamsurface Vortex Visualization 10

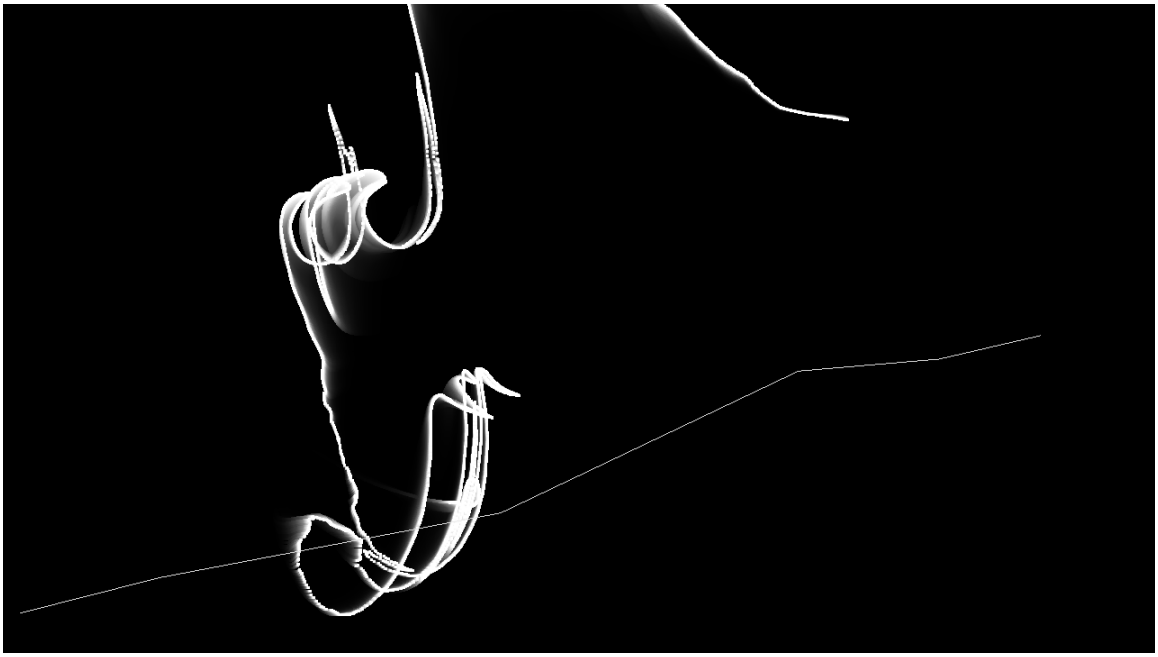

Figure 6.24: Streamsurface Vortex Visualization 11

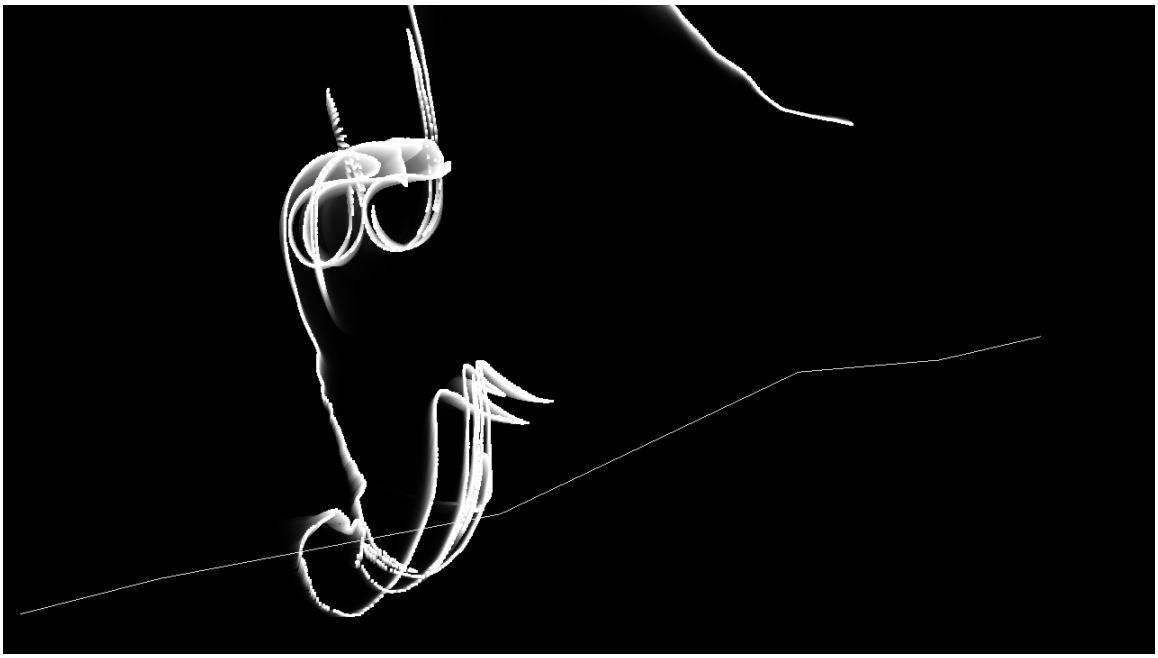

Figure 6.25: Streamsurface Vortex Visualization 12

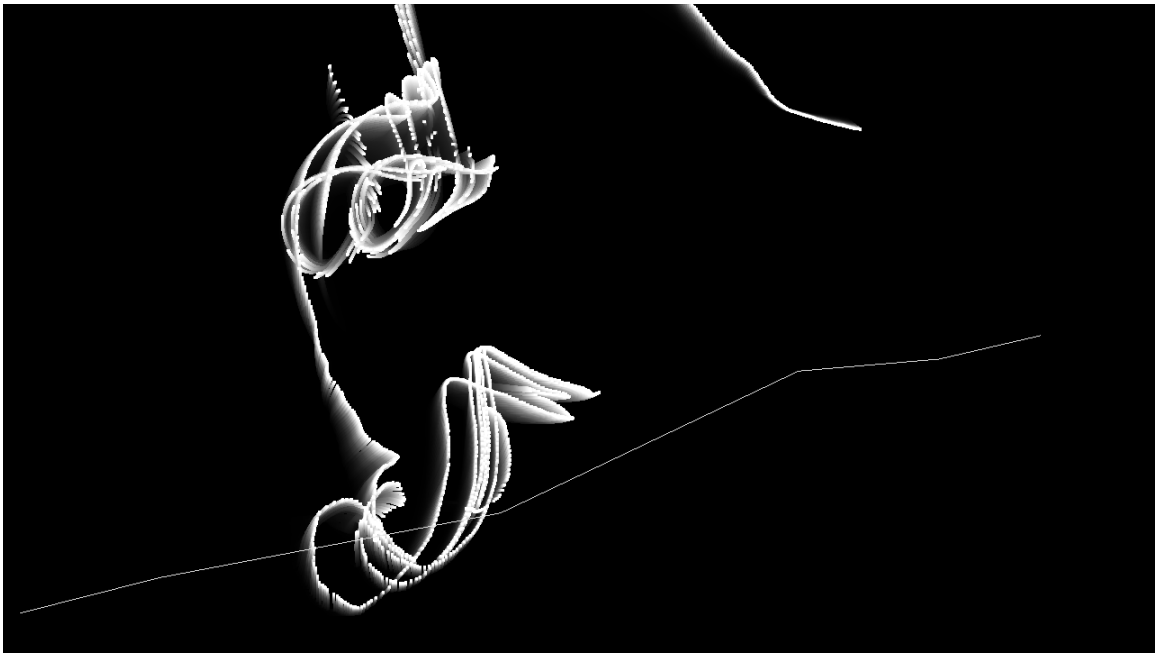

Figure 6.26: Streamsurface Vortex Visualization 13

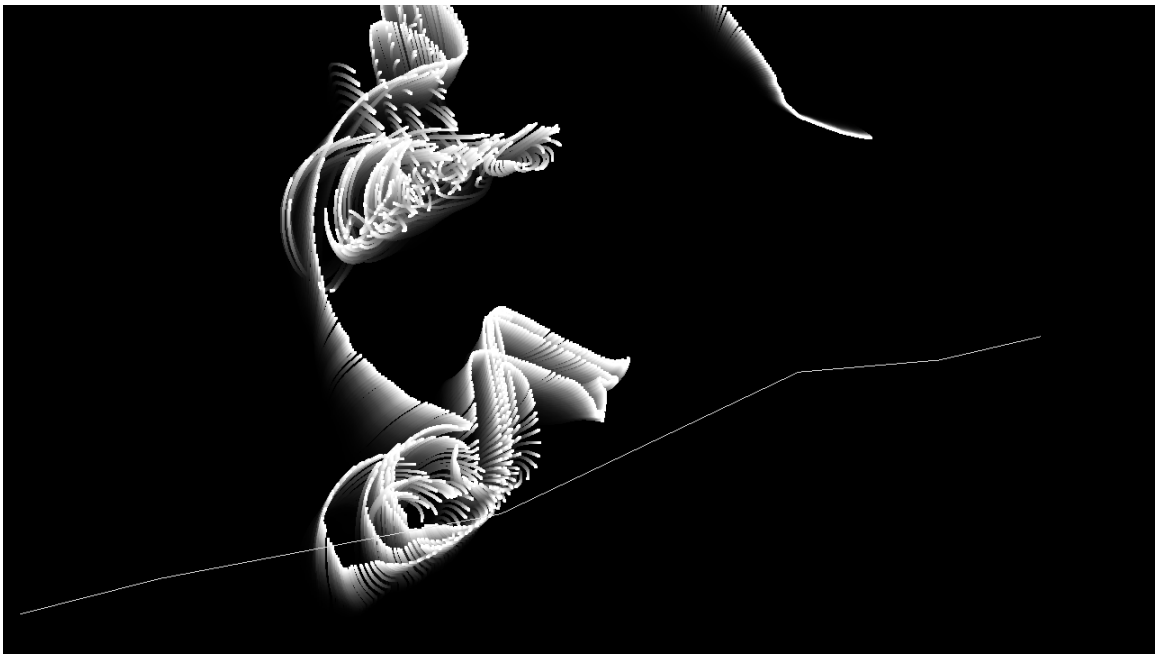

Figure 6.27: Streamsurface Vortex Visualization 14

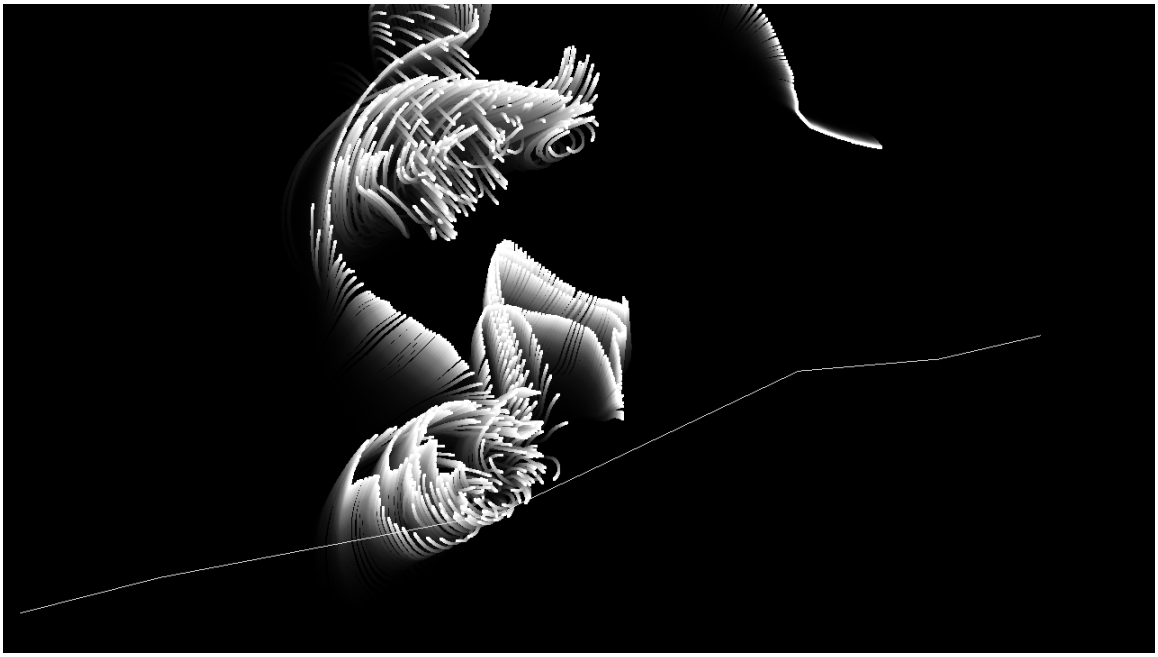

Figure 6.28: Streamsurface Vortex Visualization 15

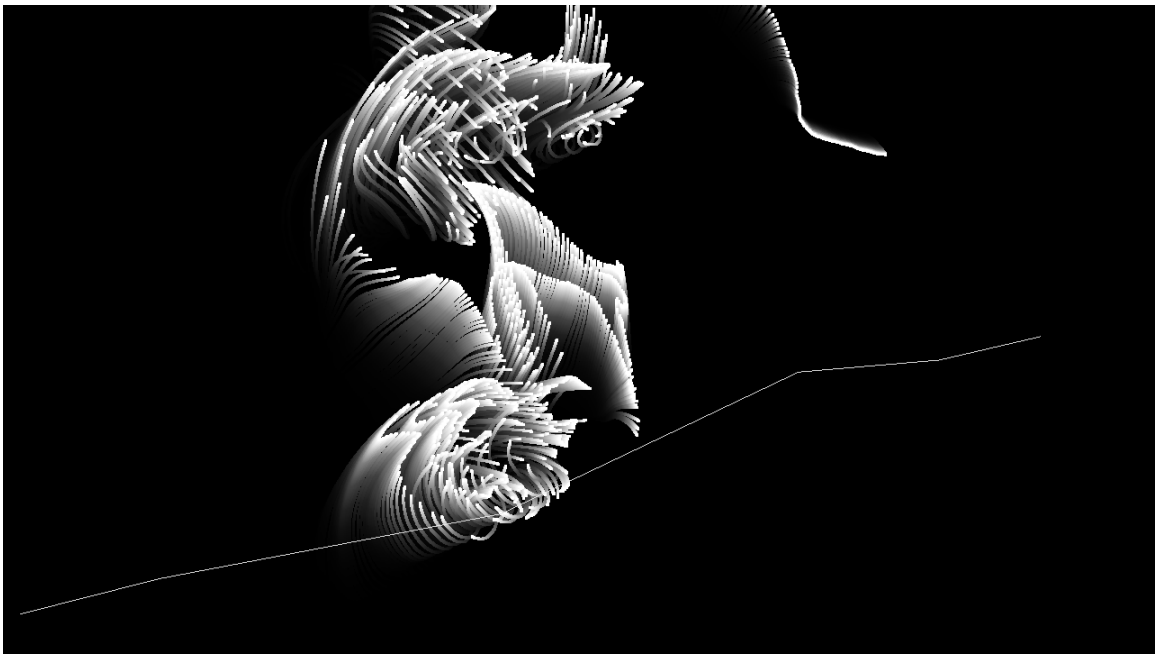

Figure 6.29: Streamsurface Vortex Visualization 16

Future Work

One possibility for extending this work is to adapt the FBO fade affect to OpenGL ES so it can run on mobile devices. This is not quite possible with OpenGL ES 2.0 since it lacks floating point textures [31]. However, if the floating point texture extension is available in OpenGL ES 2.0 or if OpenGL ES 3.0 is available [32], this visualization could run on mobile or embedded devices. The only major limitation for mobile devices is the memory constraints. To get around this problem that data could be reduced to a smaller, more focused on the area containing the wings.

Bibliography

[1] Telea, A. C. (2008) Data Visualization Principles and Practice, A K Peters, Ltd., Wellesly, MA 02482.

[2] Bryson, S. and Levit, C. (1991) In Proceedings of the 2nd conference on Visualization '91 VIS '91 Los Alamitos, CA, USA: IEEE Computer Society Press. pp. 17–24.

[3] Hultquist, J. P. M. (1992) In Proceedings of the 3rd conference on Visualization '92 VIS '92 Los Alamitos, CA, USA: IEEE Computer Society Press. pp. 171–178.

[4] Cabral, B. and Leedom, L. C. (1993) In Proceedings of the 20th annual conference on Computer graphics and interactive techniques SIGGRAPH '93 New York, NY, USA: ACM. pp. 263–270.

[5] Shen, H.-W. and Kao, D. L. (1997) In Proceedings of the 8th conference on Visualization '97 VIS '97 Los Alamitos, CA, USA: IEEE Computer Society Press. pp. 317–ff.

[6] Verma, V., Kao, D., and Pang, A. (1999) In Proceedings of the conference on Visualization '99: celebrating ten years VIS '99 Los Alamitos, CA, USA: IEEE Computer Society Press. pp. 341–348.

[7] Laramee, R. S., Weiskopf, D., Schneider, J., and Hauser, H. (2004) In Proceedings of the conference on Visualization '04 VIS '04 Washington, DC, USA: IEEE Computer Society. pp. 51–58.

[8] Schafhitzel, T., Tejada, E., Weiskopf, D., and Ertl, T. (2007) In Proceedings of Graphics Interface 2007 GI '07 New York, NY, USA: ACM. pp. 289–296.

[9] Schneider, D., Reich, W., Wiebel, A., and Scheuermann, G. (2010) In Proceedings of the 12th Eurographics / IEEE - VGTC conference on Visualization EuroVis'10 Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. pp. 1153–1161.

[10] Grabner, M. and Laramee, R. S. (2005) In Proceedings of the 21st spring conference on Computer graphics SCCG '05 New York, NY, USA: ACM. pp. 77–84.

[11] Camp, D., Childs, H., Garth, C., Pugmire, D., and Joy, K. oct. 2012 In Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on : pp. 39 –47.

[12] Haeberli, P. and Akeley, K. September 1990 *SIGGRAPH Comput. Graph.* **24(4)**, 309–318.

[13] Winner, S., Kelley, M., Pease, B., Rivard, B., and Yen, A. (1997) In Proceedings of the 24th annual conference on Computer graphics and interactive techniques SIGGRAPH '97 New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.. pp. 307–316.

[14] Gribel, C. J., Doggett, M., and Akenine-Möller, T. (2010) In Proceedings of the Conference on High Performance Graphics HPG '10 Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. pp. 163–172.

[15] Angelidis, A., Neyret, F., Singh, K., and Nowrouzezahrai, D. (2006) In Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation SCA '06 Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. pp. 25–32.

[16] Koehler, C. M. Visualization of Complex Unsteady 3D Flow: Flowing Seed Points and Dynamically Evolving Seed Curves with Applications to Vortex Visualization in CFD Simulations of Ultra Low Reynolds Number Insect Flight PhD thesis Wright State University Dayton, OH, USA (2010).

[17] Kitware, I. Vtk file formats http://www.vtk.org/VTK/img/file-formats.pdf (2010).

[18] Chapra, S. C. and Canale, R. P. (2002) Numerical Methods for Engineers with Software and Programming Applications, McGraw Hill, New York, NY 10020.

[19] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992) Numerical recipes in C (2nd ed.): the art of scientific computing, Cambridge University Press, New York, NY, USA.

[20] Fehlberg, E. Low-order classical runge-kutta formulas with stepsize control and their application to some heat transfer problems NASA Technical Report 315 NASA Washington, D.C., USA (1969).

[21] Cash, J. R. and Karp, A. H. September 1990 *ACM Trans. Math. Softw.* **16(3)**, 201–222.

[22] Group, K. The khronos group http://www.khronos.org/ (2012).

[23] Shreiner, D. and Group, T. K. O. A. W. (2009) OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1, Addison-Wesley Professional, 7th edition.

[24] Segal, M. and Akeley, K. The opengl graphics system: A specification version 3.0 http://www.opengl.org/registry/doc/glspec30.20080811.pdf (2008).

[25] Segal, M. and Akeley, K. The opengl graphics system: A specification version 2.0 http://www.opengl.org/registry/doc/glspec20.20041022.pdf (2004).

[26] Segal, M. and Akeley, K. The opengl graphics system: A specification version 3.3 core profile http://www.opengl.org/registry/doc/glspec33.core.20100311.pdf (2010).

[27] Wright, R. S., Haemel, N., Sellers, G., and Lipchak, B. (2010) OpenGL SuperBible: Comprehensive Tutorial and Reference, Addison-Wesley Professional, 5th edition.

[28] Wolff, D. (2011) OpenGL 4.0 Shading Language Cookbook, Packt Publishing, Birmingham, B27 6PA, UK.

[29] Devices, A. M. Compiler options quick reference guide, amd opteron interlagos http://developer.amd.com/Assets/CompilerOptQuickRef-62004200.pdf (2011).

[30] Devices, A. M. Amd opteron 6200 series processors linux tuning guide http://developer.amd.com/wordpress/media/2012/10/51803A_OpteronLinuxTuningGuide_SCREEN.pdf (2012).

[31] Munshi, A. and Leech, J. Opengl es common profile specification version 2.0.25 http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf (2010).

[32] Lipchak, B. Opengl es version 3.0.1 http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.1.pdf (2013).

Appendix A

Seed Line Picking Program

A program was created to generate streamlines for the streamsurface program. This program loaded the 3D models for the wings and loaded the cell sizes for all of the cells in the data file. The wings are drawn in white along with the wireframe for the currently selected cell in magenta. Lines are also drawn for the axis at the currently selected cell in red. The streamline that is being created is drawn as a green line. When the streamline is finished the file is automatically saved in the current directory with a date and timestamp. The velocity vectors, data file indicies, data file position, and camera position are also displayed. All of this is shown in Figures A.1 and A.2.

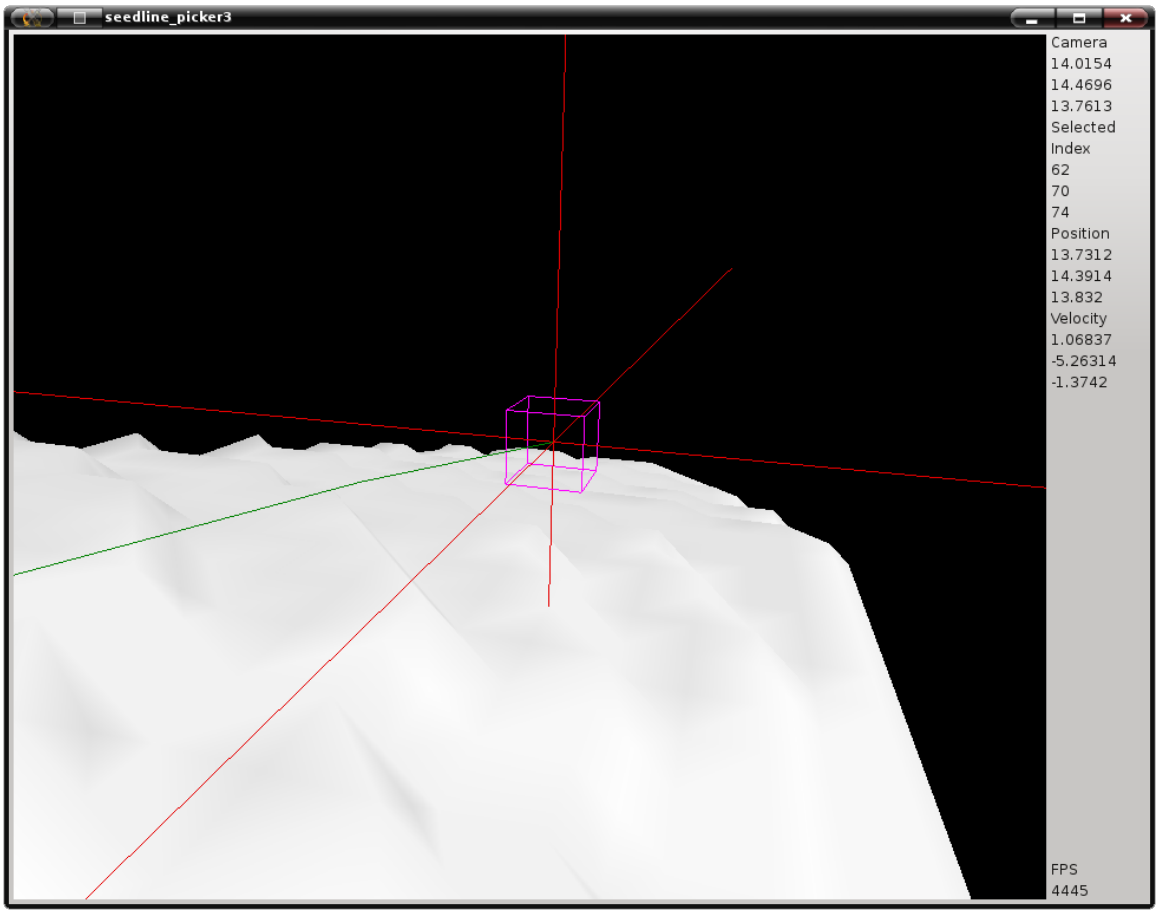

Figure A.1: Seed Line Picking Program Drawing a Streamline

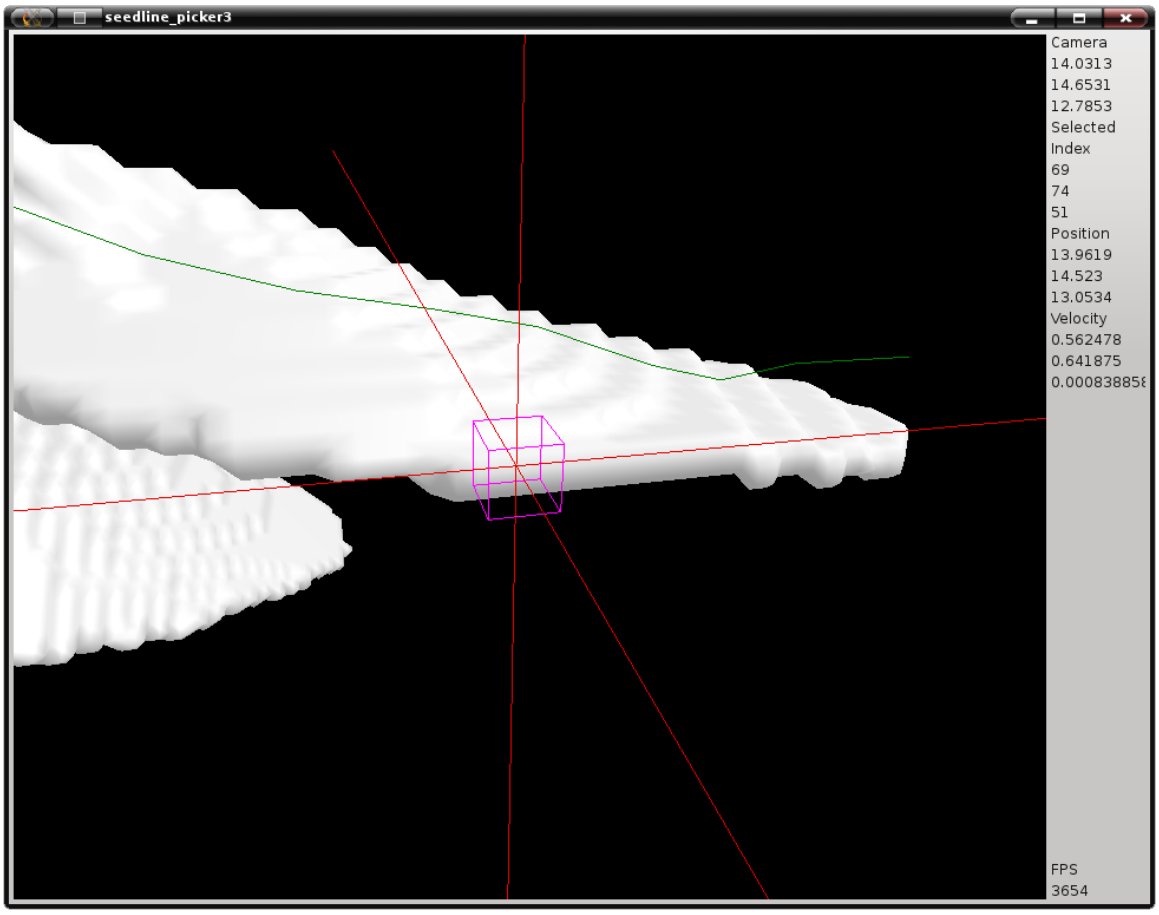

Figure A.2: Seed Line Picking Program Done Drawing a Streamline