AN ABSTRACT OF THE THESIS OF

Vasumathi Lakshmanan for the degree of Master of Science in Computer Science presented on September 21, 2006.
Title: Interactive 3D Line Integral Convolution on the GPU.

Abstract approved:

Mike Bailey

The Line Integral Convolution (LIC) is a mainstay of flow visualization. It is, however, computationally intensive, which limits its interactivity. Also, when used to view three-dimensional (3D) vector fields, the resulting images are dense and cluttered, making it difficult to perceive the flow on the interior parts of the field. This thesis describes research to make the 3D LIC more interactive by implementing it on the Graphics Processor Unit (GPU). It also includes methods to improve the clarity of the 3D LIC images. The volume dataset and a 3D noise volume are placed in GPU memory as 3D textures. The GPU is then used to perform the LIC computations and display the resulting volume. This allows the user to dynamically adjust LIC parameters and derive more insight into the 3D flow field. Various techniques such as introduction of sparsity and the use of stereographics help to de-clutter the scene. Resulting images and timing benchmarks are included.

Interactive 3D Line Integral Convolution on the GPU

by

Vasumathi Lakshmanan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 21, 2006
Commencement June 2007

Master of Science thesis of Vasumathi Lakshmanan presented on September 21, 2006.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Vasumathi Lakshmanan, Author

ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Mike Bailey for his guidance, encouragement and motivation throughout my graduate study and for providing me the opportunity to work in my area of interest. I thank Dr. Ron Metoyer and Dr. Eugene Zhang, whose courses have helped me in gaining considerable insight in my field of study, for consenting to be on my committee. I thank Dr. John Dilles for accepting to be on my committee.

I would like to thank my fellow students in the Visualization and Graphics group for their valuable suggestions and comments and especially my friend Avneet Sandhu for her support. I extend my gratitude to the EECS department for providing financial assistance through the course of my study.

I thank my parents for their love and constant support. I would like to thank my brother and sister-in-law who have guided and helped me throughout my course of study here at OSU. My special thanks to all my friends here and back in India for their support. I am grateful to have had the grace of the Almighty which enabled me to do everything.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

INTERACTIVE 3D LINE INTEGRAL CONVOLUTION ON THE GPU

1. INTRODUCTION

Vector Field Visualization has been an active area of research in scientific visualization. Vector fields are differentiated from scalar fields in that they have both magnitude and directional information. Vector field visualization is needed to visualize and hence to better understand flow patterns. It has wide spread applications in areas including scientific visualization, computational fluid dynamics and even artistic domains. Figure 1.1 shows a two-dimensional circular flow field. In this simple visualization, the directional information is encoded using the arrow heads. Scalar information like the magnitude of the field is encoded with color as well as with the length of the arrows. Thus areas in the center part of the field have low magnitude. The magnitude of the field increases with distance from the center of the field.

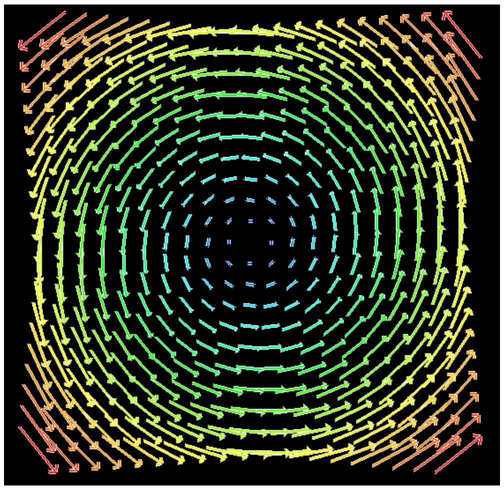


Figure 1.1: *A 2D circular vector field*

A major application area of vector field visualization is in computational fluid dynamics. Fluid flow plays an important role in several fields of study such as in determining the air flow around the wings of an aircraft, representing the interaction of molecules, studying the flow of fluids through pipes, depicting weather patterns, traffic flow, portraying the blood flow through arteries etc. In these cases the vector field is the velocity field of the element under study. Visualization of these vector fields, which have directional information, is crucial for understanding and analyzing the flow characteristics, such as convergence and divergence, curl and vorticity.

Techniques for vector field visualization can be grouped into three categories: (1) icon based-techniques (an example is shown in Figure 1.1), (2) particle tracing-based techniques, and (3) texture-based techniques. The first two methods tend to be dependent on the initial placement of icons and seed points respectively. They are likely to miss important details in the field as well. Icon-based techniques also clutter the screen.

At present, texture-based methods are a popular means of vector field visualization. These methods give dense and detailed stream-lined images. Streamlines are curves that are everywhere tangential to the flow. The dense representation of texture-based methods has the advantage that important features of the vector field can be better maintained. One of the well-known texture-based methods used for visualizing vector fields is Line Integral Convolution (LIC) [1]. It takes an input texture image and a vector field and filters the input texture along local streamlines defined by the vector field to produce an output image. The output image clearly shows almost all aspects of a vector field like sinks, sources, vortices and discontinuities. Thus, LIC is not dependent on the placement of streamlines or seed points. However we have to note that since LIC is dependant on the sampling rate, features can still be missed. Figure 1.2 shows the LIC representation of the circular vector field shown in Figure 1.1.

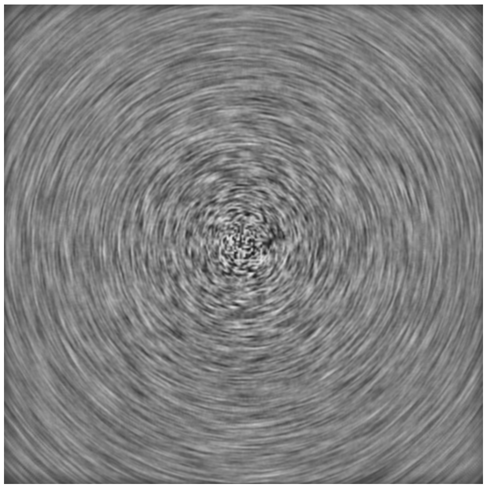


Figure 1.2: *A 2D circular field mapped on to a texture*

LIC has been widely used for many purposes including the following:

- Image processing to produce painterly effects of images by using the edge field of the image as the underlying vector field [1]
- Motion blurring using variable length LIC [1]
- Wind velocity visualization [1]
- Interactive flow visualization for an improved medical diagnosis of blood vessel malformations [2]
- 3D combustion simulation [3]
- 3D tornado simulation [3]
- Radiation therapy treatment planning [5] etc.

Line integral convolution enables us to visualize large and detailed vector fields in a reasonable display area. It selectively blurs a reference image as a function of the vector field to be drawn. LIC is widely used for viewing 2D vector fields. However, when LIC is used to visualize 3D vector fields it becomes difficult to view the interior of the vector field i.e. the vector values in the interior of the volume domain. The field is cluttered and dense. As a result of this, the inner part of the field remains hidden. Also LIC is computationally expensive in 3D. Hence the use of LIC for 3D vector field visualization has always been limited [6]. Figure 1.3 shows a 3D vector field mapped using LIC. The vector field that is mapped describes a flow around a corner. The dense nature of the texture occludes most of the details of the vector field inside the cube.

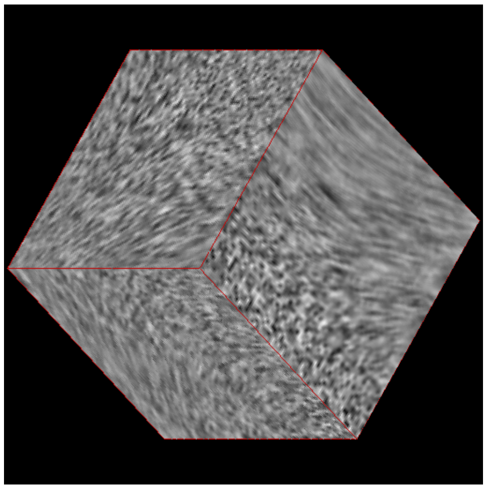


Figure 1.3: *A 3D field mapped using LIC*

This thesis focuses on ways to enhance the clarity of the images of 3D flow using 3D LIC. The implementation of 3D LIC is done on the GPU. We have implemented various techniques that would allow the user to explore different parts of the field interactively. Several methods are employed to introduce sparsity into the field so that the user can peer through the field and gain a good impression of the inner details of the field that are otherwise hidden. Figure 1.4 shows a 3D LIC image that we have obtained as a result of applying our techniques. We have used the same field that is mapped in Figure 1.3. The vector field is more discernible now, especially when it can be interactively manipulated on the screen.

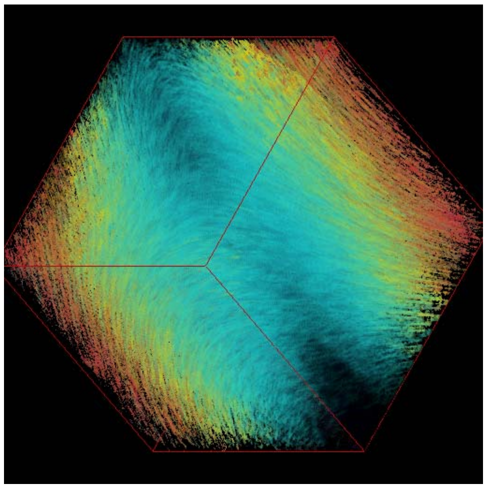


Figure 1.4: *A clearer 3D LIC image*

We use OpenGL shaders to implement 3D LIC. This greatly enhances the speed of rendering by removing computational burden from the CPU and by delegating the workload to the Graphics Processor Unit (GPU). Our user interface allows the user to explore the 3D vector field at interactive rates. The implementation is done in C++ with OpenGL as the API. GPU programming is done using the OpenGL Shading Language.

This thesis discusses the various features of our implementation and is organized into several chapters. The second chapter describes the related work done by earlier authors on vector field visualization, texture based methods in vector field visualization and in particular Line Integral Convolution. The third chapter provides a detailed description of the LIC algorithm in 2D and its implementation details on the GPU. The fourth chapter discusses our implementation of the LIC algorithm in 3D. It also provides an insight into our volume rendering approach. The next chapter presents the techniques that we have used to improve the clarity of 3D LIC images and thus overcome some of the limitations of 3D LIC. The sixth chapter dwells on the results of our implementations and observations. The final chapter concludes by providing some avenues for future enhancements.

2. PREVIOUS WORK

Some of the early methods to visualize vector fields include icon-based methods, particle traces and streamlines. Icon-based methods make use of 3D objects like arrows that are aligned with the vector field at a point [31]. The color of the object is used to encode scalar information like the magnitude of the vector field at that point. However these methods tend to clutter the screen and hence they are mostly used to visualize small vector fields (i.e. fields with less density).

Particle methods trace out the path of a weightless particle according to the vector field [31]. Suppose the initial position of the particle, i.e. the seed point, is given by (x(0),y(0),z(0)), then the aim is to find out how the path (x(t),y(t),z(t)) evolves over time. This method is also known as particle advection. The positions of the particles along a path are animated to give a sense of flow through the field.

Streamlines are lines that are everywhere tangential to the flow [18]. They can be rendered as lines or as thin flat ribbons or as tubes. Streamlines are good for showing the flow direction. However, particle traces and streamlines have the disadvantage that they may miss some eddies or currents in the vector field. They are heavily dependent on the placement of seed points or streamlines respectively.

Texture-based techniques are currently gaining popularity because they consider all the information provided by the data and give dense and detailed representations of the vector field. They are independent of the initial placement of seed points and they capture all details of the field as well.

One of the prominent methods of flow visualization is spot noise texture synthesis [7]. Spot noise texture is generated as weighted and randomly positioned spots. Variations of spot size, width, shape etc. lead to local control of the texture. A strong relationship exists between the features of the spot and the features of the texture. When used for vector field visualization, the spots are elliptically stretched along a line tangential to the vector field direction. However, if the ellipse major axis exceeds the local length scale of the vector field, the spot noise will inaccurately represent the vector field.

Image Based Flow Visualization (IBFV) [8] is one of the well-known texture-based methods for vector field visualization. This method distorts a mesh at each time-step according to the flow at that instance of time. Then it maps the image obtained from the previous iteration to the distorted mesh. The distorted image is then blended with a new noise image and this becomes the image for the current iteration. Blending helps to keep the distortion within the viewport. This method gives smooth animations of pathlines and it handles unsteady fields as well.

One of the popular texture-based techniques is Line Integral Convolution (LIC). The LIC algorithm, proposed by Brian Cabral and Leith (Casey) Leedom [1], is a modification of the DDA convolution algorithm. In the DDA convolution algorithm, a DDA generated filter kernel is defined as a line tangential to the vector field at a given point and extending for a fixed distance L in the positive and the negative direction. The pixels in the input texture are then filtered along this kernel to produce the output image. The main defect of this algorithm is that the output is heavily dependent on the shape of the filter. Also, if the length of the kernel is larger than the local radius of curvature at a point, then minute variations in the field are missed. The LIC algorithm described in [1] defines the filter kernel to be the local streamline at a point. Hence small details in the vector field are not missed by the LIC algorithm. We should note here that the same modification that was applied to the DDA algorithm can be applied to the spot noise algorithm that was discussed earlier. But the algorithm to stretch the spots along the local streamline direction is more complex and expensive than LIC.

However, the LIC algorithm is quite expensive, since to calculate the intensity at each point, all points on the streamline that passes through it are accessed. A method for increasing the speed of LIC by reducing the number of streamline computations is described in [4]. The method is based on the fact that adjacent pixels on the same streamline need to access almost the same pixels to perform integration. Recalculation of the streamline is avoided. This method makes LIC resolution independent and provides smooth texture animations.

Also, when LIC is applied to 3D vector fields, the parts of the vector field at the farther end of the user are obscured by the parts of the vector field near the user. This is due to the dense nature of the texture based methods. To overcome such limitations, many solutions have been proposed. The use of transfer functions and clipping mechanisms for the interactive exploration of 3D LIC images is discussed in [2]. This method defines a region of interest (ROI) to explore a part of the field alone. However once it is specified the ROI is fixed and cannot be changed. Two approaches to animation are also used. The first one uses a precomputed 3D LIC texture that is animated by color look-up tables. The second approach uses time volumes to interactively clip the 3D LIC volume.

The use of dye advection to highlight local features in the flow is explained in [3]. LIC is used for the global vector field view. When dye is introduced at a point, LIC smears the dye across the local streamline at the point. To get a sense of flow direction, only points in the downstream direction that correspond to cells whose negative streamlines pass through the dyed areas are colored. So to easily locate such cells, the method makes use of a flow back directed graph. A ‘bivariate’ volume rendering technique is used to display the propagation of the dye through the volume. Both of the methods described above provide means of visualizing the local features of the flow. Yet, the streamlines in the 3D flow are still cluttered together and obstruct each other. Depth information is also obscure.

Several ways to indicate the presence of depth discontinuities in 3D LIC images are proposed in [6]. It uses a sparse input noise texture to reduce cluttering of streamlines. It also uses color coding of streamlines and visibility impeding halos for the streamlines. To display the halos, two noise textures are used; one with larger scan converted spots. When LIC is performed on both the textures and when the resulting images are combined, each streamline in the second texture becomes the halo of the corresponding streamline in the first texture. The halos are rendered in the following manner. The number of times a ray has already hit a halo is kept track of. The opacity of a halo is then reduced as a function of the number of times the ray has encountered a halo. Thus halos of streamlines that are closer to the user are brighter. This method is not very efficient in cases where the streamlines are short and lack continuity information. The use of LIC to illustrate the surface shape is described in [5]. The vector field (i.e. the tensor field) in this case is taken to be the principal direction. In this method, a sparse input texture is used. When the sparse texture is advected strokes are obtained. The stroke width, length, color etc. are used to encode some scalar information. In [17], a multi-pass approach is introduced to visualize the anisotropy in symmetric 2D and 3D tensor fields. The method is similar to LIC except that instead of using noise texture values along the streamline, noise texture values in the vicinity of the streamlines are also used [17].

Several methods employ the capabilities of the underlying hardware to speed up the visualization of 3D vector fields. A means of applying lighting to the streamlines using the texture mapping capabilities of the underlying graphics hardware is provided in [16]. Reflection on streamlines increases the perception of depth. Transparency is used as a means of differentiating between the forward and the backward direction of the streamlines. The placement of streamlines in regions of interest is guided by statistical methods. A method is discussed in [10] for texture advection using the graphics hardware capabilities. It makes use of texture maps, pixel textures, hardware frame buffers etc. The disadvantage of this method is that hardware resources like buffers are limited in number.

Unsteady flows are visualized using the GPU in [11]. It uses Image Based Flow Visualization (IBFV) as the underlying algorithm for both 2D and 3D visualizations. Texture advection is performed on the GPU in [12]. On every iteration, a single slice of the 3D texture is advected. Texture based volume rendering is used to display the final result. An implementation of LIC on hardware is described in [9]. It implements LIC using pixel textures. To implement LIC using pixel textures, the noise values are provided in a luminance texture. The vector field is encoded using two textures; one for the positive component and the other for the negative component of the field. For each integration step, the pixels are accessed from the frame buffer and appropriate mapping is done. Numerical integration, interpolation of the vector field and the input noise field are all done taking advantage of the texture mapping capabilities of the hardware.

Thus methods have been employed to access the local features of the LIC images, to enhance the depth relationships of streamlines in 3D LIC and to perform LIC on the hardware. However, the problem of cluttering of streamlines and the inability to view the 3D vector field clearly has not been completely solved.

In this thesis we use different techniques and propose ways to improve the display of 3D LIC images. We focus on implementing 3D LIC using GLSL GPU programming and on ways to improve the clarity of the 3D LIC images, exploiting the capabilities of the GPU. The use of shaders for 3D LIC improves the performance of the LIC process. Also it allows for faster interactive exploration of the 3D LIC flow volume.

3. 2D LINE INTEGRAL CONVOLUTION (LIC)

In this chapter, we discuss our implementation of 2D LIC on the Graphics Processor Unit (GPU). Before discussing the implementation details of 2D LIC on the GPU, we provide an overview of the OpenGL Shading Language.

3.1 Basics of the OpenGL Shading Language (GLSL)

GLSL stands for the GL Shading Language [23]. It is a high level programming language that includes many features of the C programming language. It incorporates many features of the C++ programming language as well. It has been created by the OpenGL ARB as a means of providing programmers access to the lowest level graphics hardware.

GLSL allows programmability in the vertex and the fragment processing parts of the pipeline. Figure 3.1 shows the fixed functionality graphics pipeline and the highlighted regions indicate the functionalities that are programmable.

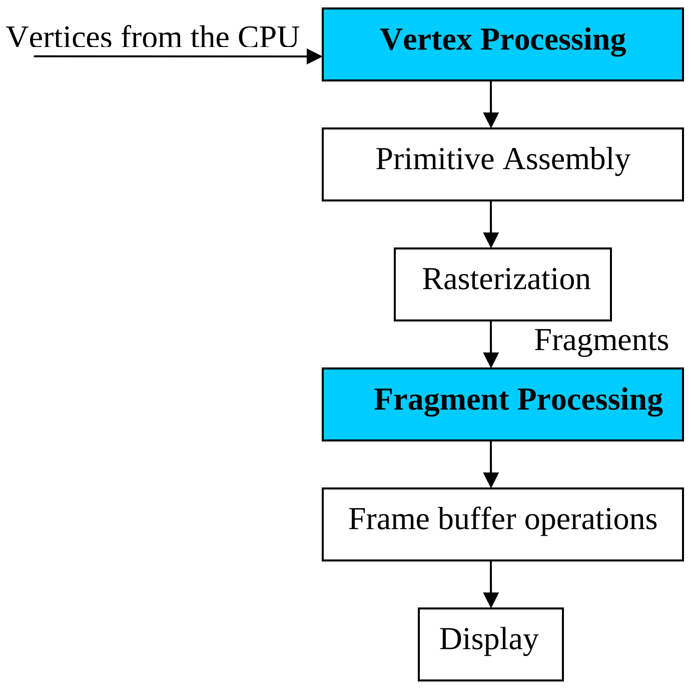


Figure 3.1: *Graphics pipeline*

The vertex processor replaces the following functionality of the OpenGL pipeline:

- Vertex transformations
- Normal normalization
- Normal transformations
- Texture coordinate generation and transformation
- Per vertex lighting

The fragment shader replaces the following functionality of the OpenGL pipeline:

- Color computation
- Fog
- Texture application
- Normal computation for per-pixel lighting

If a vertex (or a fragment) processor is used to implement one of the above mentioned functions, then the vertex (or fragment) processor should be used to implement all the functions that it replaces.

The datatypes supported by GLSL include int, float, bool, vectors, matrices, arrays, structures and samplers for texture access. For the purpose of communication, GLSL provides four types of variable qualifiers. The qualifiers and their purpose are shown below.

Table 3.1: *GLSL Qualifiers*

Qualifiers	*Purpose*
Attribute	Used to pass frequently changing per vertex information from the application to the vertex shader. They are read-only variables in the vertex shader.
Uniform	Used to pass infrequently changing information from the application to the vertex and the fragment shaders. They are also read-only variables in both the shaders.
Varying	Used to pass interpolated data from the vertex shader to the fragment shader. Varying variables are written in the vertex shader but are read-only in the fragment shaders.
Const	Used to indicate compile time constants.

3.2 The LIC algorithm

A 2D vector field is mapped on to a 2D image by the following process:

- Each output pixel is obtained as the sum of the values of the pixels in equally spaced intervals along the local streamline originating at that point and moving in both directions [1].

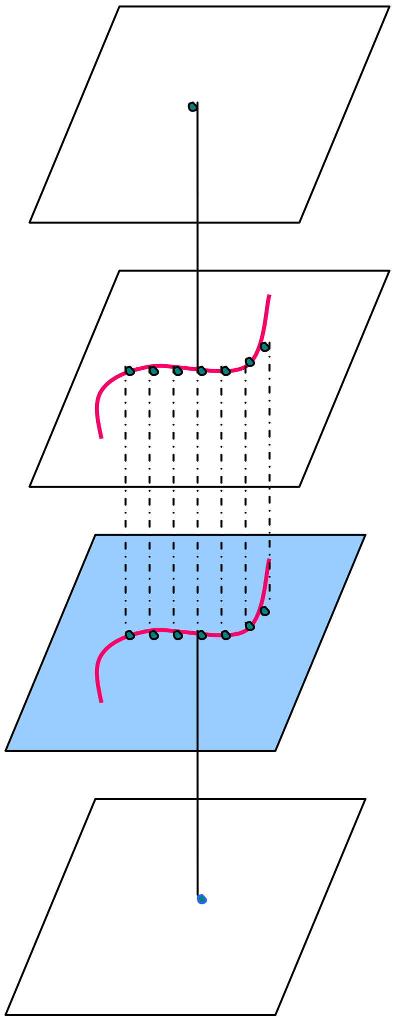
- The sum is normalized so that the intensity of the pixels remains constant over the image.
- Thus, given a point p in the input vector field, the local streamline at that point is taken as the convolution kernel K(p).
- The corresponding output pixel F’(p) is obtained as the weighted sum of the values of the input image F along K(p). The discretized version of this equation can be given as

  $$F'(p) = \frac{\sum_{i=-L}^{L} F(K(i))h(i)}{\sum_{i=-L}^{L} h(i)}$$

  where L is the user defined distance for which we move in the positive and the negative direction along the local streamline.

We have made use of a noise texture as our input image. This is because a noisy image is highly uncorrelated without patterns and hence when a vector field is mapped to it, the vector field is seen clearly. We discuss noise in more detail later in this chapter.

Figure 3.2 describes the basic operation of Line Integral Convolution. It shows how LIC is applied to a single pixel on the image. The same procedure is applied to all the pixels in the image. At the end, the noise texture is squished and stretched along the local streamlines of the vector field.



- a pixel on the input noise texture.

- the local streamline (defined by the input vector field) that originates at the pixel and goes for a user-defined distance in the positive and negative directions. The green dots show the pixels along the streamline.

- the pixels in the input image corresponding to the positions on the local streamline are accessed.

- the value of the pixel in the final image is obtained as the normalized sum of the values along the local streamline. The values are weighted as a function of distance.

Figure 3.2: *A single step of the LIC process*

3.3 LIC Implementation on the GPU

3.3.1 Vertex Shader

In the vertex shader, the texture coordinates for the vertex are computed and stored in the varying variable gl_TexCoord[i], where i indicates the multi texturing level. The texture parameters and the texture wrapping modes are defined by the application itself. The function ftransform() transforms the incoming vertex by the ModelView and the Projection matrices and the final transformed vertex position is written to the special output variable gl_Position.

```
void main(void)
{
        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_TexCoord[1] = gl_MultiTexCoord1;
        gl_Position = ftransform();
}
```

Figure 3.3: *Vertex shader performing 2D LIC*

3.3.2 Fragment Shader

Both the input texture (ImageTexture) and the vector field (VectorTexture) are passed as uniform variables from the application as texture samplers. The user defined length of convolution and other user defined parameters are also passed as uniform variables. The texture coordinates for the incoming fragment are obtained using *vec2 st = gl_TexCoord[0].st*. The texture values of the incoming fragment are obtained using the *texture2D* function. This function takes the texture unit and the texture coordinates (s and t) as arguments and returns the corresponding texel value. The vector field at that position is obtained as *vec2 v = vec2(texture2D(VectorTexture, st))*. The value of the input image corresponding to that position is obtained as *vec3 color = vec3(texture2D(ImageTexture, st))*. We then move along the local streamline in the positive (negative) direction using $st = st \pm v$. At each point the corresponding color of the input image is added to the net color. The normalized sum of the colors at the points that we visit is the final color of the fragment and it is assigned to the special built-in variable gl_FragColor.

```
uniform sampler2D ImageTexture;
uniform sampler2D VectorTexture;
uniform int Length;
uniform float TwoOverRes;
uniform float OneOverNum;
uniform int Bias;
int i;
vec2 v;
int LengthP;
void main( void )
{
        vec2 st = gl_TexCoord[0].st;
        v = vec2( texture2D( VectorTexture, st ) );
        v -= vec2(.5,.5);
        v *= TwoOverRes;
        vec3 color = texture2D( ImageTexture, st );
        st = gl_TexCoord[0].st;
        LengthP = Length - Bias;
        for(i=0;i<LengthP;i++)
        {
                st -= v;
                st = clamp( st, 0., 1. );
                color += texture2D( ImageTexture, st );
                v = vec2( texture2D( VectorTexture, st ) );
                v -= vec2(.5,.5);
                v *= TwoOverRes;
        }
        st = gl_TexCoord[0].st;
        LengthP = Length + Bias;
        for(i=0;i<LengthP;i++)
        {
                st += v;
                st = clamp( st, 0., 1. );
                color += texture2D( ImageTexture, st );
                v = vec2( texture2D( VectorTexture, st ) );
                v -= vec2(.5,.5);
                v *= TwoOverRes;
        }
        color *= OneOverNum;
        gl_FragColor = color;
}
```

Figure 3.4: *Fragment shader performing 2D LIC*

3.3.3 User Interface and Parameters

Our user interface for 2D LIC is created using the Graphics User Interface Library (GLUI) [25]. Range sliders are provided to let the user vary parameters like the length and bias. Length indicates the number of steps we take from a point, in the positive and the negative direction along the local streamline at that point. Bias is used to provide an animation effect. It is used to increase (and decrease) the distance for which we move in the positive (and negative) direction. Hence as the user moves the slider corresponding to this parameter, the vector field has the effect of moving. We have tested our implementation using a circular vector field.

3.4 Noise

Noise is used as the input image. Some of the desired characteristics of noise are that noise needs to be continuous and repeatable and yet give the appearance of randomness. This means that the function that is used to create noise should be able to provide the same output value for a given input. This is essential in cases where we need to draw an object in different angles or when we want to draw the same object in an animation sequence [23]. If noise is not repeatable, the object would look different every time it is drawn. But noise should not have regular patterns.

Since a noise image is highly uncorrelated, when a vector field is mapped to it, the streamlines are more clearly seen. Our input noise image is created using a positional noise function.

3.4.1 Positional Noise

Positional noise depends on the placement of random numbers and so it may not be well distributed in the given range (Figure 3.5). But it is simple to generate and serves our purposes well.

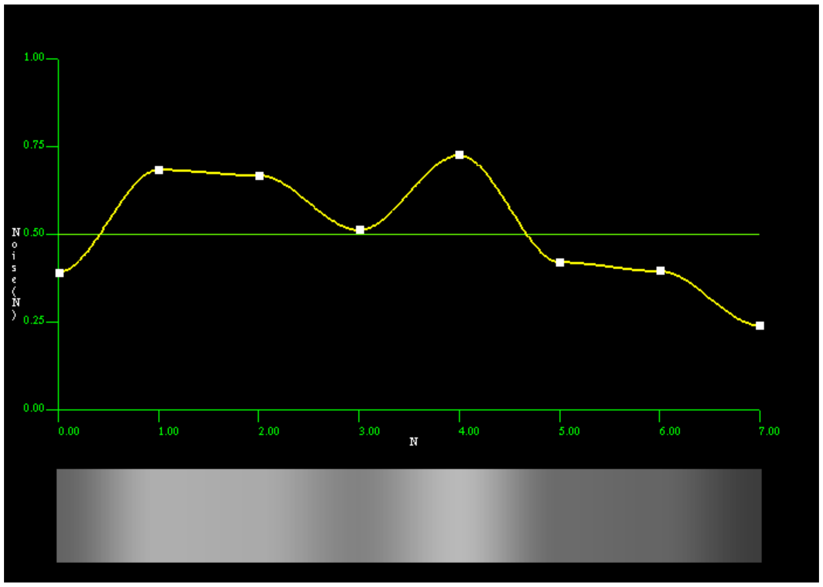


Figure 3.5: *Positional noise*

We have used a circular vector field to test our implementation of 2D LIC. We have tested with simple images as well as with the noise images created. (Figure 3.6, 3.7).

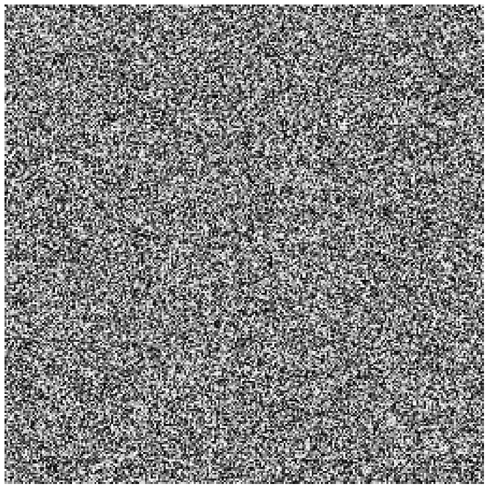


Figure 3.6: 2D *noise*

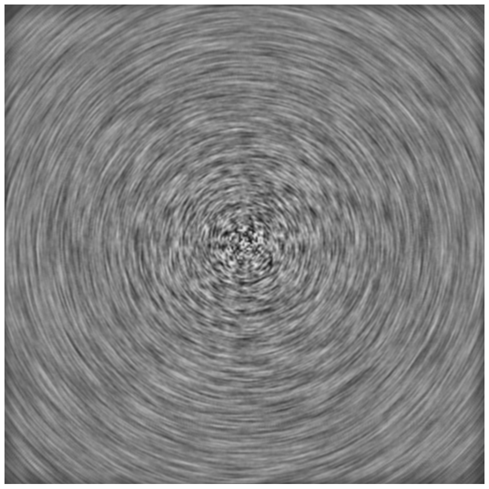


Figure 3.7: *2D LIC - A circular vector field*

3.5 Time Varying Fields

LIC has been traditionally used for viewing static vector fields. We have experimented with the use of LIC for time varying 2D vector fields as well. In order to do this, we have made use of a simple vector field defined as:

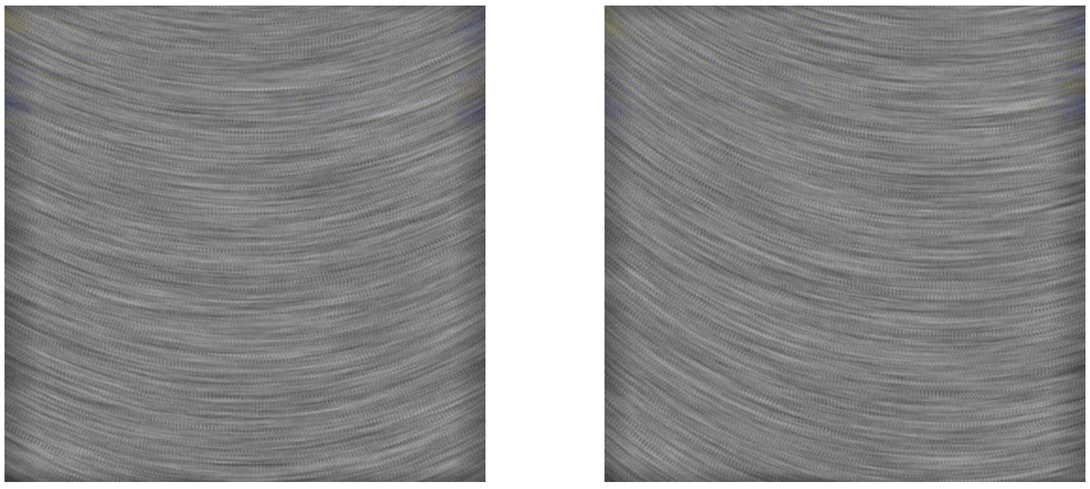
$$V\,x = \cos(s - (2 * Time));$$
$$V\,y = \sin(s - (2 * Time));$$

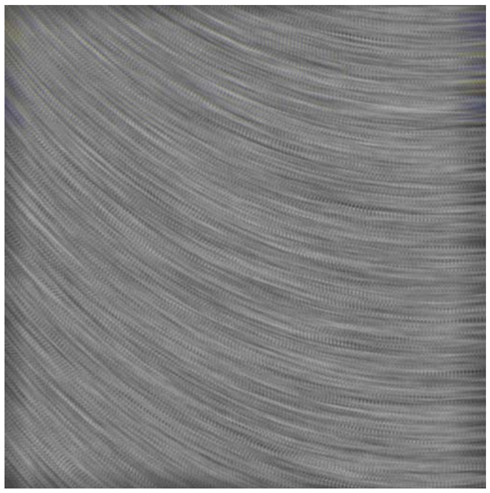
where $V\,x$ is the horizontal component of velocity

$V\,y$ is the vertical component of velocity

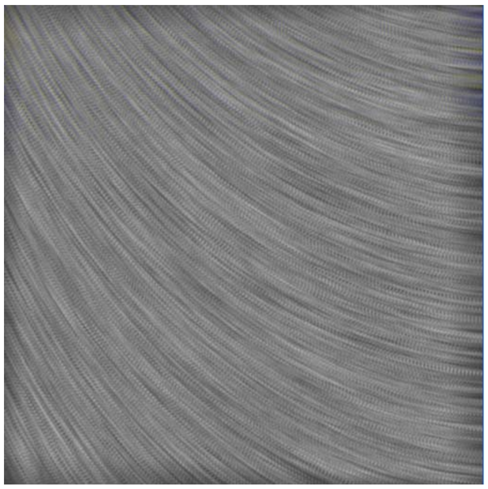
s is the s value of the texture coordinate.

The Figure 3.8 shows the field at different times. In our user interface we have a slider that controls the value of time. As the user moves the slider, the field moves showing the vector field at different times.

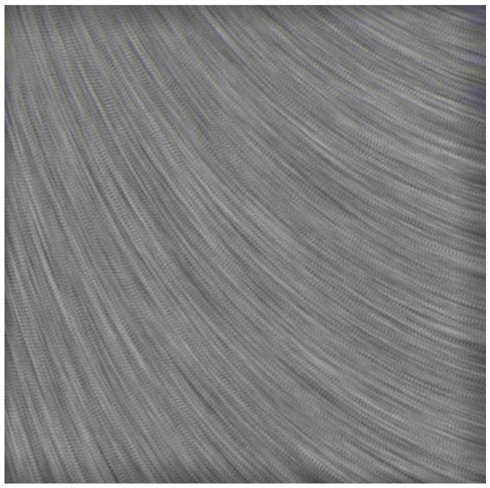

(a) *(b)*



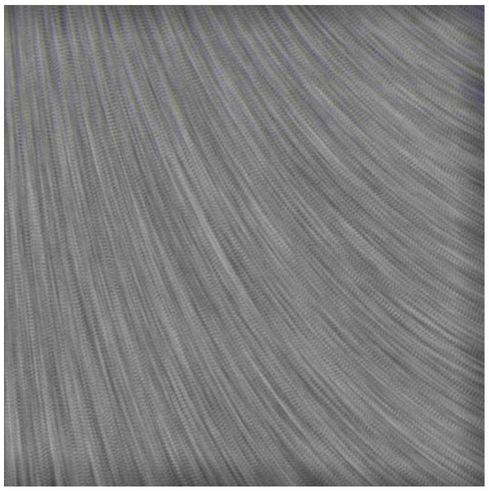
(c)



(d)



(e)



(f)

Figure 3.8: *Time varying vector field (a) Time = 0 (b) Time = .1 (c) Time = .2 (d) Time = .3 (e) Time = .4 (f) Time = .5*

4. 3D LINE INTEGRAL CONVOLUTION

The process of mapping a 3D vector field to a 3D texture image is the same as discussed for mapping a 2D vector field to a 2D texture image. The input noise texture is blurred locally by using the local streamline at a given point as the convolution kernel at that point. The local streamline originates at the given point and goes in the positive and the negative direction for a user defined distance. In the case of 3D LIC, the local streamline is in three dimensions. Hence in the output image, the intensity values are related to the vector field's flow direction.

4.1 Volume Rendering

4.1.1 3D Noise generation

3D LIC requires volumetric data to which the 3D vector field can be mapped. In 3D LIC, the vector field is mapped to a 3D texture image. The 3D input texture image is a 3D noise data set. We create our volumetric noise data as an RGBA volume. The RGBA volume is described as a 3D four-vector data set. The R, G and B values correspond to the Red, Green and the Blue components of the noise. In our case, we create grayscale noise and hence these values are the same for a given point. A stands for the Alpha/Opacity value that controls the transparency. We create the noise using positional noise as described in the previous chapter. The 3D noise data set is written to a file and is read into a 1D unsigned character array in the application.

4.1.2 Volume rendering approach

We now describe the method that we have used to render the volumetric noise data. Volume rendering refers to the process of representing a 3D scalar field data. In our case, the scalar field is the noise data. Volume rendering techniques do not make use of intermediate geometric representations.

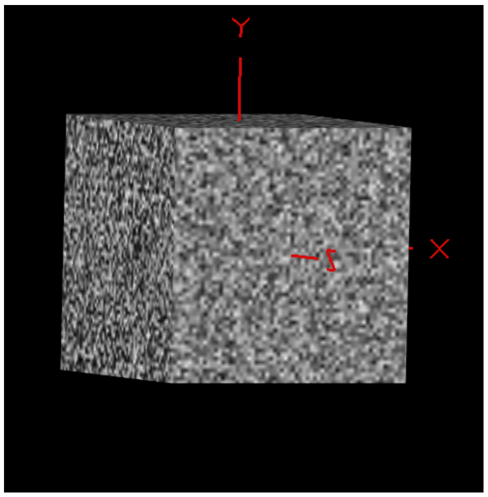
Our volume rendering approach can be described as follows:

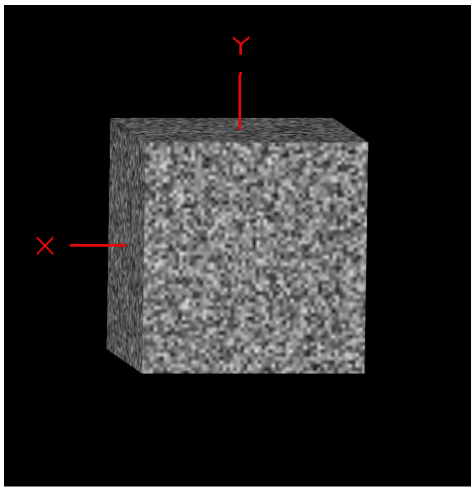
1 The volume is drawn plane by plane from back to front. A routine is used to determine which axis closely represents the viewing direction of the user and whether it is the positive side of the axis or the negative side of the axis.

2 Depending on the rotation of the volume and the axis that is facing the user, the parallel planes in the X, Y or the Z direction are drawn. For instance, if the Z axis is facing the user, then the XY planes along the Z direction are drawn.

3 The planes are composited from the positive to the negative direction or from the negative to the positive direction depending on whether it is the negative side of the axis or the positive side of the axis that faces the user. Thus the direction of the axis that is facing the user determines the direction of composition of the planes.

The images below show the 3D noise data rendered using the method described above. The alpha values of all the points are taken to be 1, which means that they are completely opaque. In Figure 4.1(a) the positive z axis faces the user. In Figure 4.1 (b) the negative z axis faces the user.





(a) *(b)*

Figure 4.1: *Volume Rendering (a) Positive Z direction (b) Negative Z direction*

4.2 Mapping a 3D Vector field to a Texture Image

The 3D vector field is also represented as a 3D texture. At each point in the 3D space under consideration, the x, y and the z components of the vector field are calculated using the given vector field equation. Then the values of all three components of the field, for all the points in the 3D space, are written to a file in the form of bytes. The values are initially calculated as floating point numbers. They are then converted to lie in the range from (0-255). Hence the three components of the vector field are encoded as the red, green and blue colors respectively. We do not use floating point textures since not all hardware support them. In the application program, the file is read into an unsigned character array and is passed to the fragment shader to be used in performing LIC.

4.3 3D Textures

Texture mapping is a technique that allows us to map a value from an array to a corresponding point in space. Texture maps are rectangular arrays of data that can contain color data, luminance data, alpha data etc. The individual values in a texture map are called texels [24]. The texture maps can be 1D, 2D, 3D or cube maps. In 3D texture mapping, for each point in the 3D space, we find the corresponding texel in the 3D texture map space. The 3D point is then colored based on the value of the corresponding texel.

3D Texture mapping capability was added to Version 1.2 of OpenGL. A 3D volume scene can be created from layers of parallel 2D rectangles. In memory, the rectangles are arranged in a sequence [24]. The steps involved in 3D texture mapping and the corresponding commands in the OpenGL API and GLSL are as follows [24][23]:

1 Create a texture object with the glBindTexture command.
   - glBindTexture(GL_TEXTURE_3D_EXT, VectorTextureID)

2 Specify how the texture object is accessed with the glTexParameter command. This specifies the wrapping mode which can be clamp or repeat. It also sets the type of filtering that is used during texture access, which can be linear or nearest filtering. The choice of the filtering technique affects the clarity of the output of our application. This issue is discussed in detail in a later section.
   - glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S,   GL_CLAMP)
   - glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP)
   - glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R_EXT, GL_CLAMP)
   - glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER, GL_LINEAR)

3 Specify a texture for the texture object. To specify a 3D texture we use the function glTexImage3D. There are no universal image file formats for 3D data. Hence we store our 3D data in a file of our own design and read it into a

1D unsigned character array. This array is provided as the last parameter to the function. Our noise data consists of four components – RGBA as discussed earlier. Hence the third parameter of the glTexImage3D function is set to GL_RGBA to indicate that the number of components in the texture is four. Similarly the eight parameter, which indicates the format of the pixel data is also set to GL_RGBA. The other parameters of this function indicate the resolution, level of detail etc. of the 3D texture.

- glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA, NUMX, NUMY, NUMZ, 0, GL_RGBA, GL_UNSIGNED_BYTE, VectorTex)

4 Enable texture mapping with the glEnable command with GL_TEXTURE_3D as its argument.

- glEnable(GL_TEXTURE_3D)

5 Set the active texture unit with the glActiveTexture command. The texture unit that is set by this command is the one that is accessed by texture coordinate processing commands. The texture unit itself is an underlying piece of graphics hardware that performs various texturing operations.

- glActiveTexture(GL_TEXTURE1)

6 Pass the active texture unit number to the vertex and the fragment shaders as a uniform variable.

- glUniform1i(VectorTextureLoc, 1)

The noise data as well as the vector field data are read from a file by the application as 3D textures. Each of these textures is bound to a separate texture unit and is passed to the vertex and the fragment shaders for processing.

4.4 Implementation on the GPU

We now describe the implementation of 3D LIC on the GPU using the OpenGL shading language.

4.4.1 Vertex Shader

In the vertex shader, the incoming vertices are transformed by the ModelView and the Projection matrices using the ftransform() function. The vertices are written to the special output variable gl_Position in clipped coordinates [23]. Compilers may generate an error message or the results are undefined if the vertex shader is used for vertex processing and no value is stored in gl_Position [23]. The texture coordinates are passed from the application using the attribute variable gl_MultiTexCoord0. They are stored in the varying variable gl_TexCoord[i], where i indicates the multi-texturing level used.

```
void main( void )
{
        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_TexCoord[1] = gl_MultiTexCoord1;
        gl_Position = ftransform();
}
```

Figure 4.2: *A simplified version of the vertex shader performing 3D LIC*

4.4.2 Fragment Shader

As in 2D LIC, the input noise texture (ImageTexture) and the vector field texture (VectorTexture) are passed as samplers from the application to the fragment shader. The vector field is also represented as a 3D texture. At each point, the x, y and z component of the vector field is encoded using the RGB colors of the texture. In the fragment shader, the texture coordinates for the incoming fragment are obtained using *vec3 stp = gl_TexCoord [0].stp*. The texel value is obtained using the *texture3D* function. This function takes the texture unit and the texture coordinates as arguments and returns the corresponding texel value. The vector field at that position is obtained as *vec3 v = vec3(texture3D(VectorTexture, stp)).* The value of the input image corresponding to that position is obtained as *vec4 color = texture3D(ImageTexture, stp).* Then we move along the local streamline in the positive (negative) direction using *stp = stp ±v.* At each point along the streamline, the corresponding color of the input image is added to the net value. Finally the sum is normalized. This is taken to be the final color of the fragment and is assigned to the special output variable gl_FragColor.

Two other uniform variables - TwoOverRes and OneOverNum are sent to the fragment shader from the application. TwoOverRes is defined as (2 / Resolution of the volume). The value two in the numerator is used to get the velocity range to be (-1.,-1.) to (1.,1.). It is divided by the resolution of the image to make the step size to be one pixel. OneOverNum is defined as (1/2L) where L is the user defined LIC length. It is used to normalize the sum of the colors. Rather than performing the division in the fragment shader, these values are sent from the application. This is because the division needs to be done only once this way rather than for each incoming fragment.

```
uniform sampler3D ImageTexture;
uniform sampler3D VectorTexture;
uniform int Length;
uniform float TwoOverRes;
uniform float OneOverNum;
uniform int Bias;
int i;
vec3 v;
int LengthP;

void main( void )
{
    vec3 stp = gl_TexCoord[0].stp;
    v = vec3( texture3D( VectorTexture, stp ) );
    v -= vec3(.5,.5,.5);
    v *= TwoOverRes;
    vec4 color = texture3D( ImageTexture, stp );
    stp = gl_TexCoord[0].stp;
    LengthP = Length - Bias;
    for(i=0;i<LengthP;i++)
    {
        stp -= v;
        stp = clamp( stp, 0., 1. );

        color += texture3D( ImageTexture, stp );

        v = vec3( texture3D( VectorTexture, stp ) );
        v -= vec3(.5,.5,.5);
        v *= TwoOverRes;
    }
    stp = gl_TexCoord[0].stp;
    LengthP = Length + Bias;
    for(i=0;i<LengthP;i++)
    {
        stp += v;
        stp = clamp( stp, 0., 1. );

        color += texture3D( ImageTexture, stp );

        v = vec3( texture3D( VectorTexture, stp ) );
        v -= vec3(.5,.5,.5);
        v *= TwoOverRes;
    }
    color *= OneOverNum;
    gl_FragColor = color;
}
```

Figure 4.3: *A simplified version of the fragment shader performing 3D LIC*

4.4.3 User Interface and Parameters

Our user interface for 3D LIC is created using the Graphical User interface Library. (GLUI) [25]. Length indicates the number of steps we take from a point, in the positive and the negative direction along the local streamline at that point. When length increases, the output image is blurred more in the direction of the vector field as more pixels are taken into consideration. For our application, the results are good for values of length between 7 and 12. This is because enough pixels have to be accessed on the streamline to blur the image in the direction of the streamlines of the vector field. But if the length is increased further, then it leads to over blurring. Bias is used to provide an animation effect in the same way as done for 2D. Both of these parameters are user defined and are passed to the fragment shader as uniform variables. Our user interface for 3D LIC has several other functionalities as well. Those functionalities are described in a later chapter. A snapshot of the user interface is provided in the Results chapter.

4.5 Tests with a vector field

The vector field shown in Figure 4.4 is created using an equation that describes flow around a corner [19]. The equation is described as follows:

$$V_x = -3 + 6.*x - 4.*x*(y+1.) - 4.*z$$

$$V_y = 12.*x - 4.*x*x - 12.*z + 4.*z*z$$

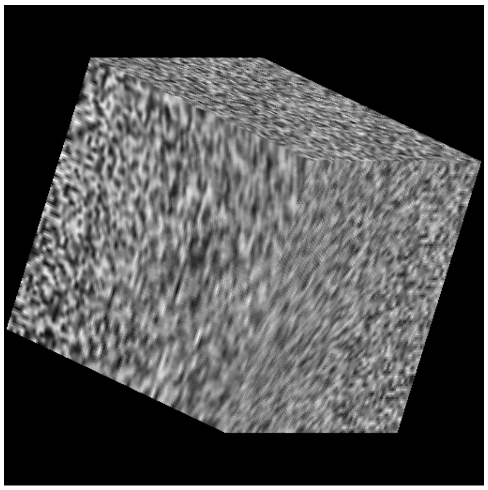
$$V_z = 3. + 4.*x - 4.*x*(y+1.) - 6.*z + 4.*(y+1.)*z$$

where V_x,V_y and V_z are the x,y,z components of the vector field.

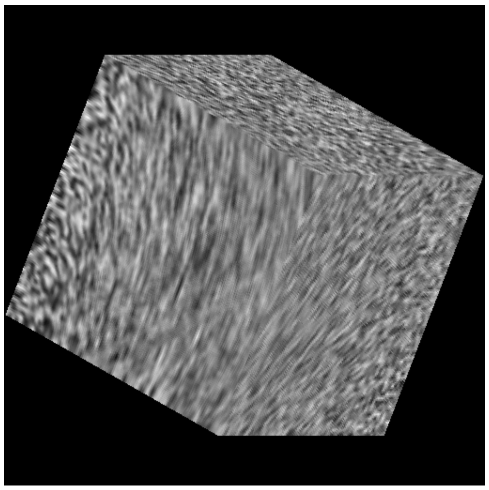
x,y,z are the coordinates of the point under consideration.

4.6 Images

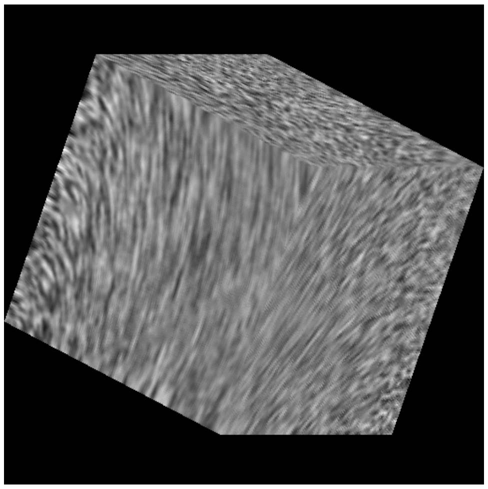
The following images show the effect of the length parameter on the output of LIC.



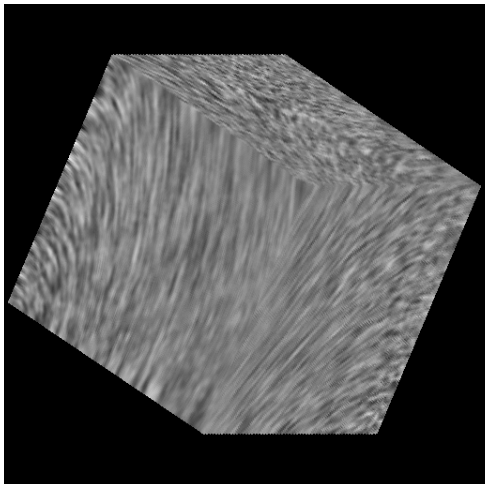
(a)



(b)



(c)



(d)

Figure 4.4: *Different lengths of LIC (a) Length = 3 (b) Length = 6 (c) Length = 10 (d) Length = 20*

5. IMPROVING THE CLARITY OF 3D LIC

The implementation of 3D LIC was discussed in the previous chapter. Some of the limitations of 3D LIC include the cluttering of streamlines, inability to efficiently perceive depth differences, the dense nature of the 3D LIC images etc. In this chapter, we discuss in detail the various methods that we have used to improve the clarity (i.e. to reduce the effects of denseness of the 3D texture and cluttering of the streamlines) of 3D LIC images. Several techniques are employed to overcome the limitations of using 3D LIC for visualizing 3D vector fields.

We have classified the techniques that we have employed into three categories:

- Methods for improving the global display of the vector field: These methods enable the user to get a good global view of the vector field. They aim at providing the user with good insight into the inner parts of the vector field as well. They also encode scalar data so that as much information as possible is conveyed to the user.
- Methods for enhancing the local display of the vector field: The goal of these methods is to allow the user to view and analyze local regions of interest in the vector field.
- Methods for improving depth perception: They increase the clarity of the spatial organization of the streamlines in the 3D vector field, thus enabling the user to distinguish near parts of the vector field from the far away parts.

5.1 Viewing the vector field globally

Texture-based methods are efficient to display 2D vector fields. But when they are used to view 3D vector fields, it can be difficult to obtain a clear global view of the field. We have used several methods to enhance the global display of 3D fields.

Particularly, the introduction of *sparsity* into the dense texture lets the user look through the field clearly. The following are the techniques employed:

5.1.1 Transparency

The opacity of each point in the 3D space is determined by the alpha value at that point. The alpha value is given by the fourth component of the RGBA vector. When the alpha value is 1, the point is completely opaque. When the alpha value is 0, the point is completely transparent. To introduce transparency, the alpha value is reduced below 1.

5.1.1.1 Alpha Blending

In Computer Graphics, transparency is obtained using the method known as Alpha Blending. The alpha value (α) actually specifies how the foreground colors are merged with the background colors when they are laid on top of one another. The equation for alpha blending is

$$\text{Color}_{\text{final}} = \alpha\,\text{Color}_{\text{foreground}} + (1-\alpha)\text{Color}_{\text{background}}$$

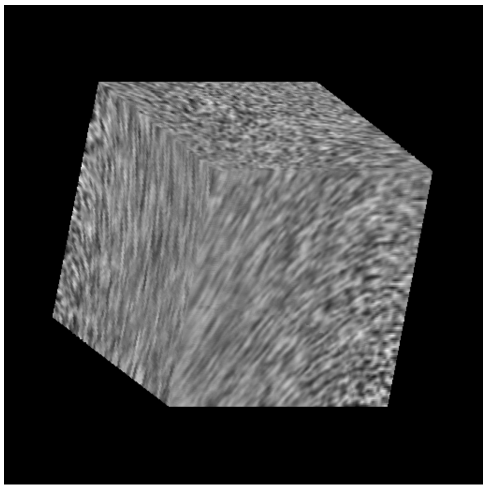
In our implementation, as discussed in chapter 3, the planes are composited on top of one another. When α has a value less than 1, the colors of the planes are blended together according to the above equation. Initially the alpha value is 1 and only the bounding planes are seen. When a lower value of alpha is assigned, the user can see through the vector field.

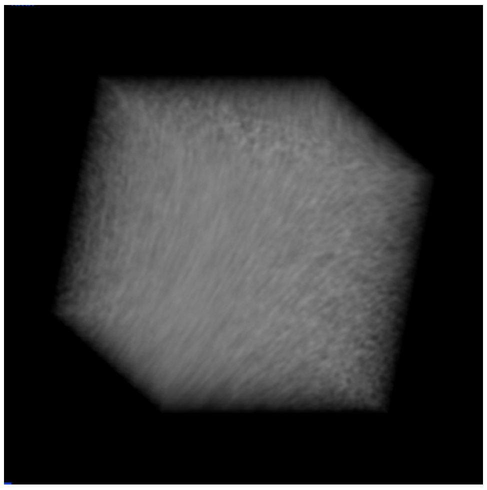
5.1.1.2 Implementation

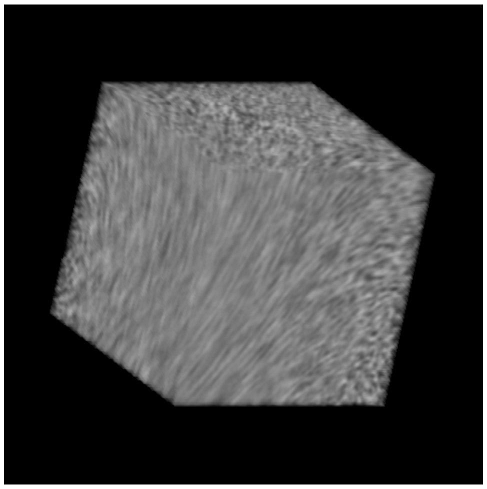
In OpenGL, alpha blending is implemented using the *glBlendFunc()* function. It defines how much of the foreground and the background colors will be used. Before using this function, blending is enabled using the *glEnable()* function with GL_BLEND as its argument.

Our implementation allows the user to interactively change the alpha value using a slider in the user interface. The alpha value as set by the user is sent as a uniform variable to the fragment shader. In the fragment shader, the alpha value that is passed as the uniform variable replaces the actual alpha value of the incoming fragment. Thus the opacity of the fragment is varied.

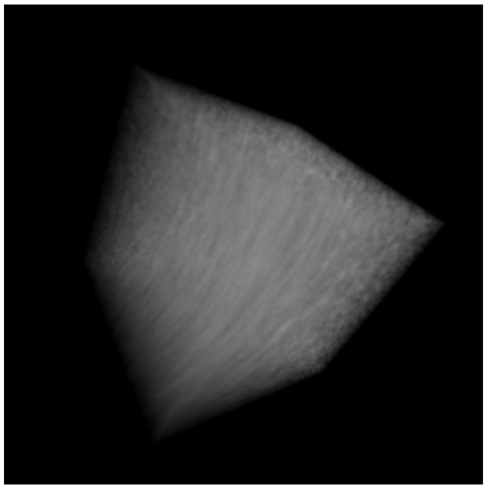
The following images show the effect of introducing transparency. We find that introducing transparency has limited usage as the vector field is still dense and details may be missed because of the transparency factor.


(a)


(b)



(c)



(d)

Figure 5.1: *Transparency (a) Alpha = 1 (b) Alpha =. 2 (c) Alpha =. 5 (d) Alpha =. 2 (Perspective View)*

5.1.2 Encoding magnitude with color

Basic Line Integral Convolution is used to view only the directional information of the vector field. It does not show information like the magnitude of the vector field. Viewing scalar information like the magnitude of the vector field is important in gaining a deep understanding of the nature of the vector field. In our implementation, we have encoded this scalar quantity using colors.

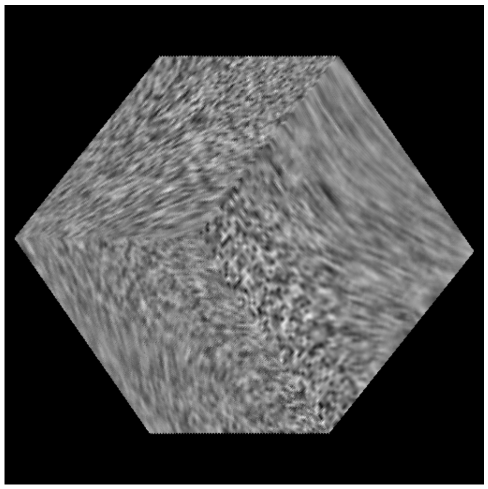
5.1.2.1 Implementation

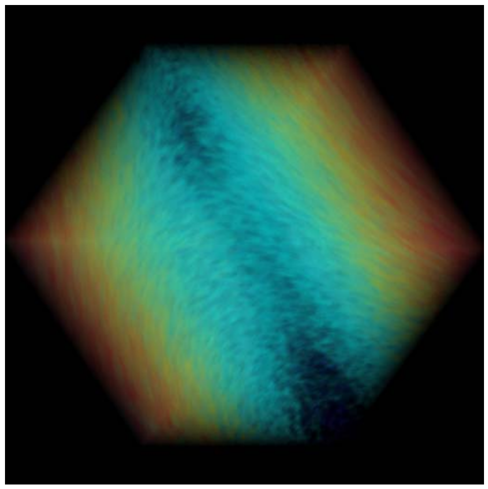
The magnitude of a vector denotes the length of the vector at a point. In other words, it shows how strong the vector field is at a particular point. Given a vector R(x_v,y_v,z_v) at a point P(x,y,z) the magnitude of the vector at P is given by

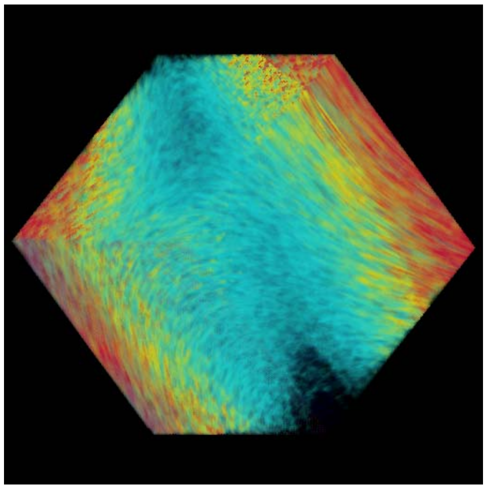
$$\sqrt{x_v{}^2 + y_v{}^2 + z_v{}^2}\ .$$

At each point in the 3D space, we find the magnitude of the vector field. Then we encode regions of larger magnitude with red color and regions of lesser magnitude with cooler shades. Hence the user is able to get an idea of the magnitude of the vector field at a particular region by looking at the colors in that region.

The following images show the effect of encoding magnitude with color in a vector field. The red regions are parts of the vector field with a very high magnitude. The blue regions are parts of the vector field with lesser magnitude. Regions colored yellow indicate the moderate parts of the vector field.


(a)


(b)



(c)

Figure 5.2: *Encoding magnitude with color (a) Without adding Color (b) With Color and Transparency (Alpha =. 2) (c) With Color (encoding Magnitude)*

5.1.3 Highlighting areas with larger magnitude

As discussed in the previous section, the presence of the portions of the vector field with low magnitude is a hindrance to viewing areas with more activity. In some applications, areas of low magnitude are not of great significance. Hence it becomes useful to highlight areas with high magnitude by reducing the opacity of the areas with low magnitude.

5.1.3.1 Implementation

We multiply the input texture's alpha value by the magnitude of the vector field. Hence we assign a low opacity value to those areas with low magnitude. Areas of the vector field with a large magnitude are highlighted, giving a clearer display.

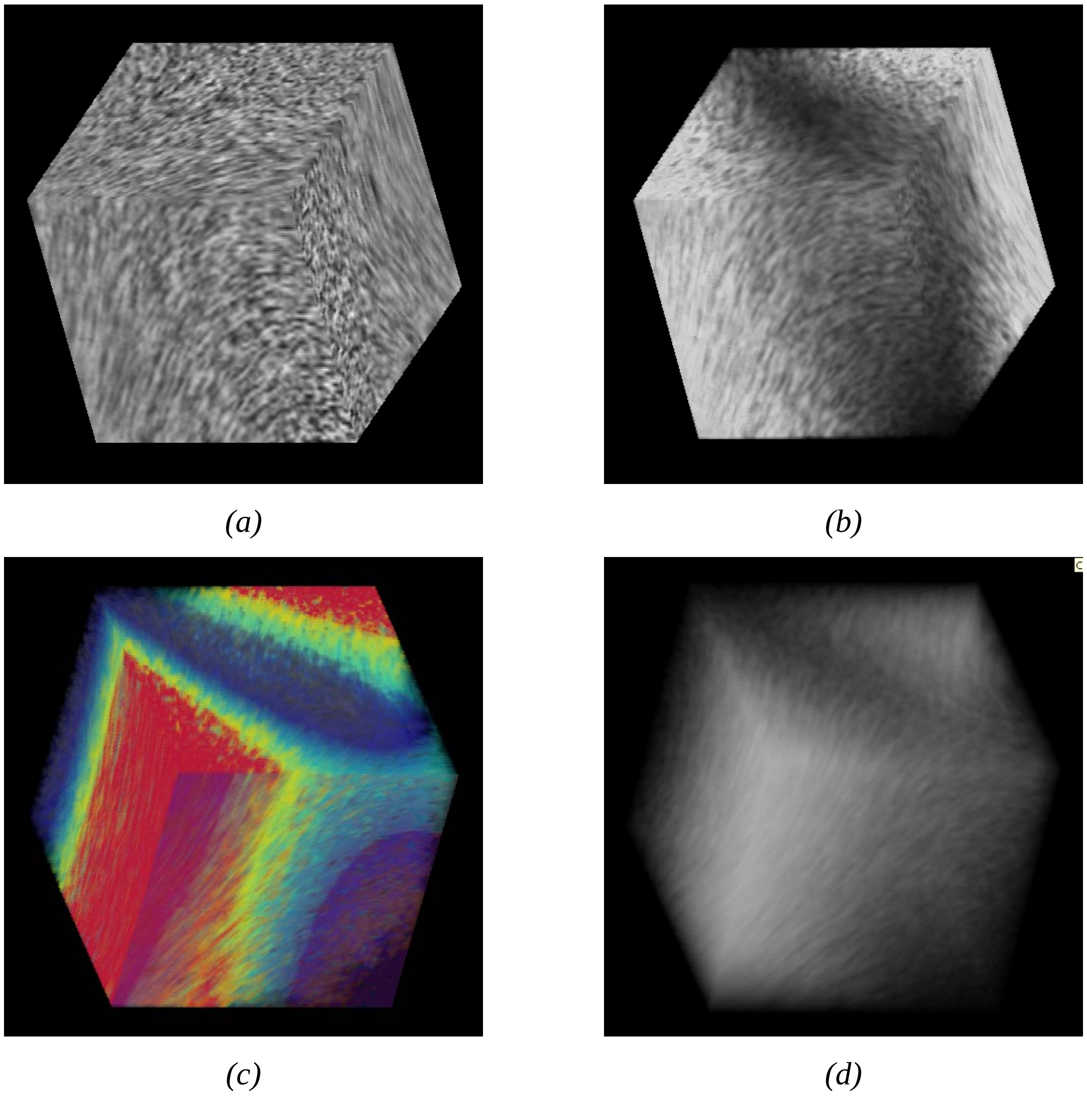


Figure 5.3: *Highlighting larger magnitude areas (a) Without Magnitude (b) With Magnitude (c) With Magnitude and Color (d) With Magnitude and Transparency*

5.1.4 Sparsity

Sparsity techniques involve strategically removing part of the detail to reveal the overall function. If sparsity is introduced, the individual streamlines are more clearly seen. This is because the user can look through the vector field and get a global view of the field.

5.1.4.1 Define a new texture image

We have introduced sparsity by defining a 3D texture “T2” with certain evenly-distributed points set to black. The rest of the points in the texture are set to white. This texture is then passed on to the shaders. In the fragment shader, every time a point is accessed in the streamline computation, a check is made to see if the corresponding point in the input texture T2 is equal to black. If so, the current fragment is discarded. A fragment is discarded using the *discard* keyword in the shader. This keyword causes the fragment shader to terminate without writing the current fragment to the frame buffer [23]. In effect all points on all streamlines that pass through the “black” point are not drawn. This introduces gaps between the streamlines, enabling the user to look through the vector field and gain knowledge about the inside of the field as well.

5.1.4.2 Sampling

The black points should have the following properties:

- They should not have a regular pattern. The sampling theory states that when the samples are regular, the coherence of the samples interfere with the coherence of the images to produce aliasing [29]. Aliasing effects appear as “jagged edges” in images when high frequencies appear as low frequencies producing regular patterns that are easy to see. In our application, if the distribution of points is too regular, then the streamlines are broken into smaller parts and there is no continuity.
- They should be as equally spaced as possible. I.e., they should be well distributed. If the points are cluttered together in one region, then in our application, the streamlines would be discarded to a large extent in that region alone. The other regions would still be dense.

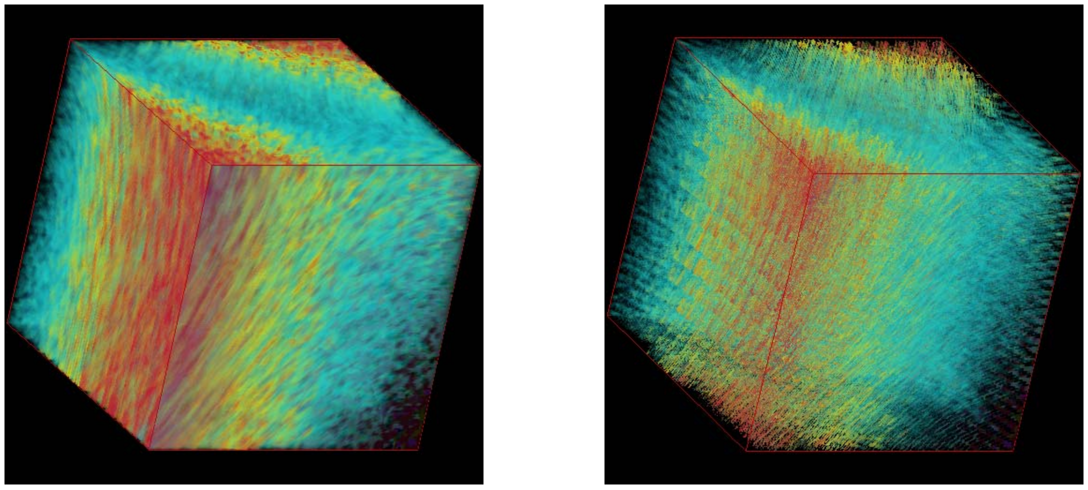
We have made use of the following sampling methods: (1) Stochastic Sampling and (2) Low-discrepancy sequences. We now discuss these methods in detail.

5.1.4.2.1 Stochastic Sampling Methods

The main idea behind stochastic sampling is that when the sampling is irregular, the higher frequencies appear as noise rather than as aliases [29]. The human visual system is more sensitive to aliases than to noise. Hence better results are obtained with stochastic sampling.

(a) Checkerboard distribution

In this method, the texture T2 is created as a 3D checkerboard pattern, with alternating black and white texels. In our implementation, the user can control the ratio of the number of black texels to white texels using a slider. However this distribution lacks randomness. It is a form of regular sampling and hence it leads to aliasing effects. This is shown in Figure 5.4.



(a) *(b)*

Figure 5.4: *Checkerboard distribution (a) Before applying Sparsity (b) With Sparsity (~ 23,250 random points)*

(b) Poisson disk distribution

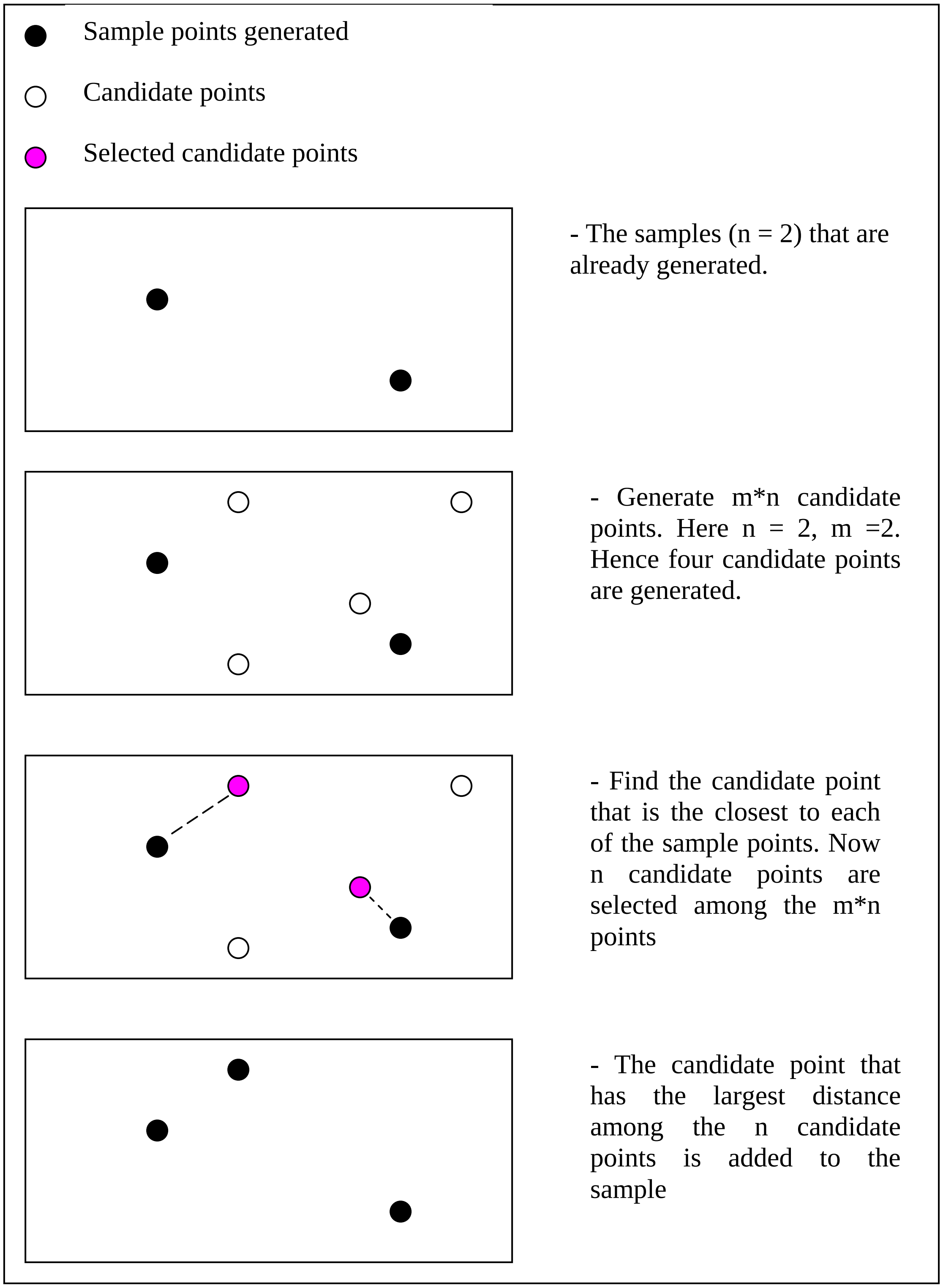
Poisson disk sampling is a generalization of Poisson Sampling. This distribution is an even-looking one. Each random value that is generated is retained only if it satisfies a minimum distance constraint. A radius around the point is defined and no two points are allowed to be closer than the radius [13]. Also, the points are placed as close as the radius will allow. This distance constraint decreases the magnitude of noise. Without the constraint the points may tend to clutter together at some areas alone. Hence this method is more effective.

However, Poisson disk distribution is a computationally expensive method. We use an approximation to the algorithm as described in [14]. This method generates m*n candidate points to generate the (n+1)th sample; where m is a constant. Hence the number of candidate points is proportional to n.

The pseudo code for the method that we have used is given as follows:

```
For every new point that is to be generated
    Generate m*n random candidate points
    For every point P that has already been generated until now
        Find the closest of the candidate points to P (use the Euclidean
        distance)
    EndFor
    Now there are n candidate points; where n is the number of points that
    are already added. Associated with each candidate point is its distance
    to the already existing point
    Among the n candidate points, find the point P' with the largest
    distance associated with it
    Add P' to the set of already generated points
EndFor
```

Figure 5.5 illustrates how the third sample point is calculated with Poisson-Disk Distribution, given the first two sample points.



● Sample points generated

○ Candidate points

● Selected candidate points

- The samples (n = 2) that are already generated.

- Generate m*n candidate points. Here n = 2, m =2. Hence four candidate points are generated.

- Find the candidate point that is the closest to each of the sample points. Now n candidate points are selected among the m*n points

- The candidate point that has the largest distance among the n candidate points is added to the sample

Figure 5.5: *Poisson disk distribution-illustration*

This method generates good sampling patterns. However, it is still an O(n^3) algorithm; where n is the number of random points to be generated. So to speed up the process, we have generated the points on a few planes and then replicated them over the rest of the planes. We have used a value of m=10 for our purposes.

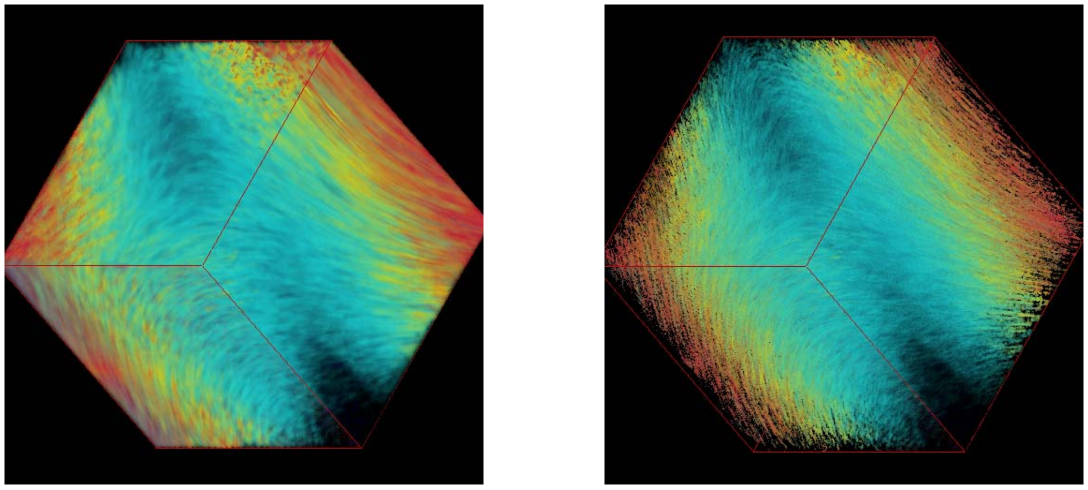

(a) *(b)*

Figure 5.6: *Poisson disk distribution (a) Before applying Sparsity (b) With Sparsity (~134,544 random points)*

Advantages

- As seen in Figure 5.6 this method introduces sparsity in the vector field with good spacing between the streamlines. This is the due to the even distribution of points.
- The streamlines are continuous. This is because of the irregular distribution of points without bunching.

Disadvantages

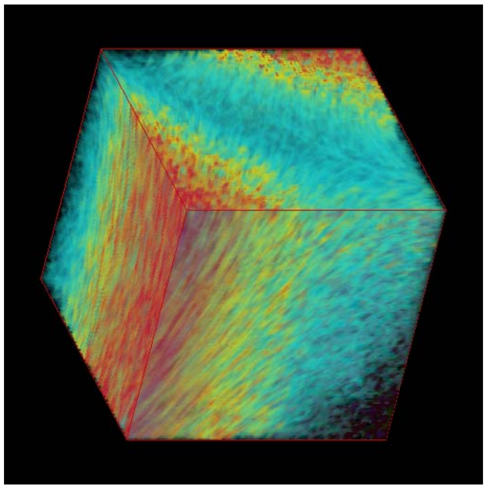
- This method is time consuming. The method that we have used cannot be used for interactive sampling.
- This method is just an approximation to the Poisson disk method. The actual method might give better results.

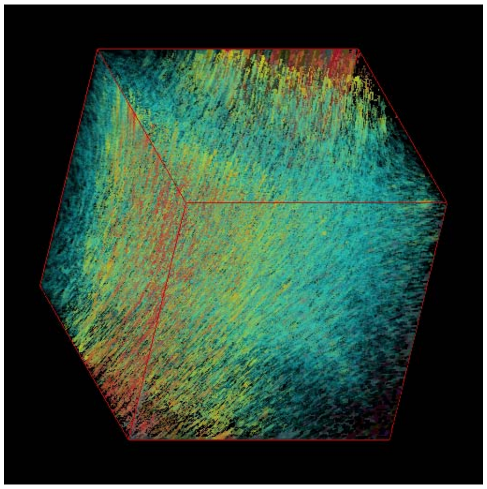
(c) Jittering

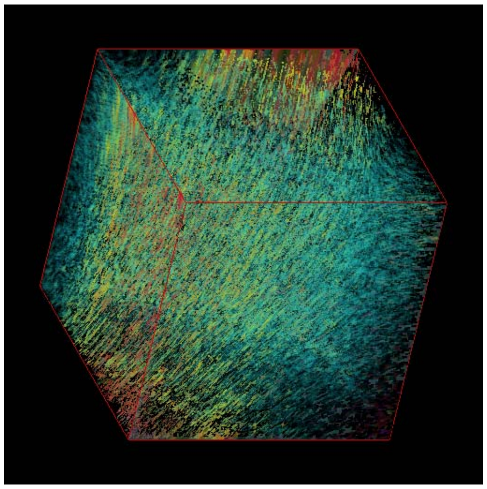
Jittering is a straightforward and a fast approximation to Poisson disk distribution [5]. This method perturbs the original points randomly. A point is set to black if its neighbors do not fall inside a user defined radius. Thus it avoids samples from bunching together at any region. We allow the radius to be user defined, and hence the user can interactively change the radius thus increasing/decreasing the sparsity.

The pseudo code for generating samples using jittering is given as follows:

```
Perturb the original points randomly
For each point P in the 3D space
        Check if any of P's neighbors fall inside the user defined radius
        If not, then add P to the set of already generated points
EndFor
```


(a)


(b)



(c)

Figure 5.7: *Jittering (a) Before applying Sparsity (b) With Sparsity (~ 373,313 random points) (c) With Sparsity (~467,754 random points) i.e.; with a larger radius than Figure 5.7 (b)*

Advantages

- This method is easier than Poisson disk distribution.
- It is straight forward.
- It works at interactive rates.

Disadvantages

- Jittering contains more low frequency energy in its spectrum [14] and hence it does not give results as good as Poisson disk distribution.
- This method does not produce samples that are as well distributed as the Poisson disk method. As a result, as seen in Figure 5.7, the spacing between the streamlines and the continuity of the streamlines is not as good as that of Poisson disk distribution.

5.1.4.2.2 Low Discrepancy Sequences

The Hammersley and the Halton point sets are low discrepancy sequences which are used for Quasi-Monte Carlo methods. The purpose of Quasi-Monte Carlo methods is to find the integral of a function as the average of the function at certain points. In Quasi-Monte Carlo methods these points are generated using low discrepancy sequences such as the ones mentioned above. In these methods, a deterministic formula generates a uniformly distributed and stochastic-looking pattern [15].

Discrepancy measures how uniformly distributed the sampled points are [15]. Low discrepancy indicates that the samples are well distributed. These low discrepancy sequences have the following property: If we consider (1/8)th of the volume, (1/8)th of the samples belong to this part of the volume. Such uniform distribution is an important requirement of our application. Hence we have used these low discrepancy sequences for generating a set of evenly distributed points in 3D space. We now discuss these methods in detail.

(a) Hammersley points

This method of sample generation generates a fixed sequence of samples that are well distributed. The general method of point generation is given as follows: The ith "d" dimensional point is given by

$$(i/N,\ \phi_{p1}(i), \phi_{p2}(i), \ldots.., \phi_{pd-1}(i))$$

where p1,p2,,….,pd-1 are any sequence of prime numbers with $p1<p2<\ldots<pd-1$ and N is the total number of samples . We have used 2 and 3 as the prime bases.

To obtain $\phi_{px}(i)$,

- Convert "i" to the representation of i in base px.
- Reflect this number about the decimal point

- Convert the reflected value to a decimal number again.

For example, the first 3D point is given by (1/250000, $\phi_2(1)$, ϕ_3 (1)), assuming that N = 250000, p1 = 2 and p2 = 3. Here i = 1.

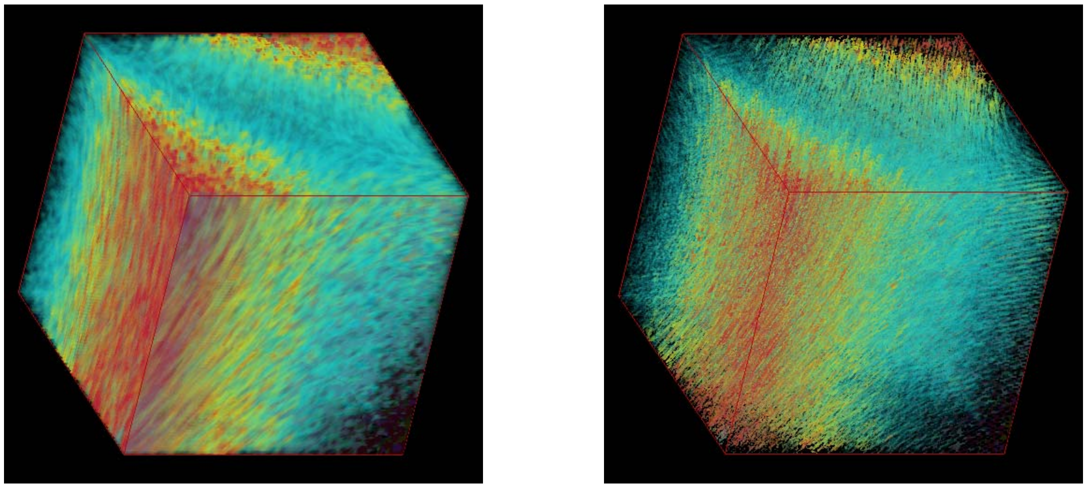
$\phi_2(1)$ is calculated as:

- The representation of 1 in base 2 is 1_2.
- After reflection about the decimal point it becomes 0.1_2.
- 0.1_2 in decimal is 0.5.

$\phi_3(1)$ is calculated as:

- The representation of 1 in base 3 is 1_3.
- After reflection about the decimal point it becomes 0.1_3.
- 0.1_3 in decimal is 0.333333.

Hence the first 3D point is (1/250000, 0.5, 0.333333) for the given assumptions.



(a) *(b)*

Figure 5.8: *Hammersley points (a) Before applying Sparsity (b) With Sparsity (~ 250,000 random points)*

Advantages

- This method is computationally fast. The user can interactively control the number of points which are assigned the black color using a slider in our user interface.
- Figure 5.8 shows that this technique gives well spaced and continuous lines. This is because of the low discrepancy of the generated points.

Disadvantages

- As the base value increases, the samples tend to become more and more regular [15].

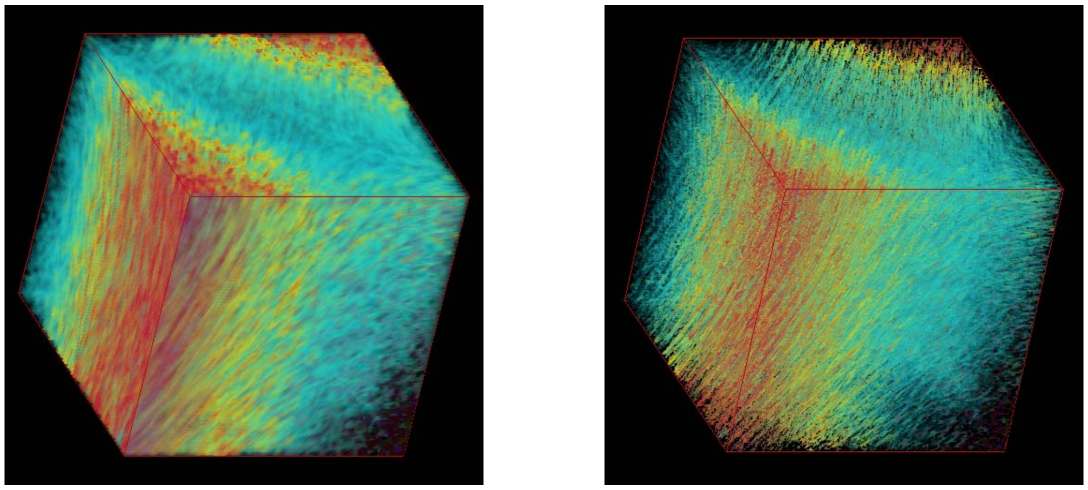
(b) Halton points

This method produces another low discrepancy sequence. In this method, the ith “d” dimensional point is given by

$$(\phi_{p1}(i), \phi_{p2}(i), \ldots.., \phi_{pd}(i))$$

where p1,p2,,….,pd are any sequence of prime numbers with $p1<p2<\ldots<pd$. We have used 2, 3 and 5 as the base values.

To obtain $\phi_{px}(i)$, we have used the same method as discussed for Hammersley point generation.



(a) *(b)*

Figure 5.9: *Halton points (a) Before applying Sparsity (b) With Sparsity (~ 250,000 random points)*

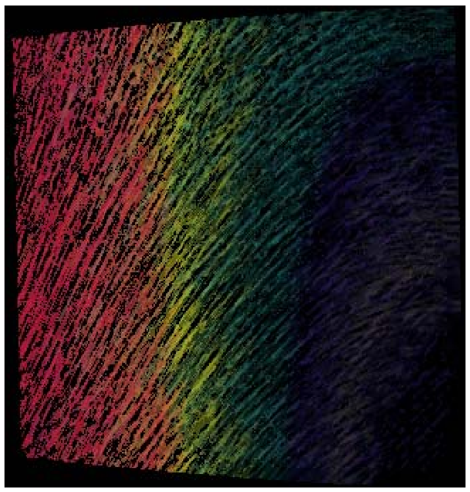
Advantages

- This method is also fast. The user can interactively change the number of points generated using a slider in our interface, thus controlling the sparsity introduced in the vector field.
- Because of the low discrepancy property of the generated point set, the streamlines are well spaced and continuous as shown in Figure 5.9.

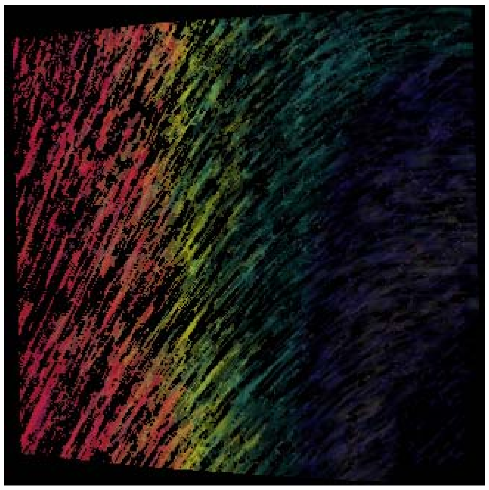
Disadvantages

- As the base value increases, the samples exhibit patterns. This is shown in the fact that Hammersley distribution gives slightly better results than Halton distribution in terms of the continuity of the streamlines. This is because of the fact that we have used 5 as the base value for the third dimension in the Halton distribution, which is higher than any of the base values used by the Hammersley distribution.

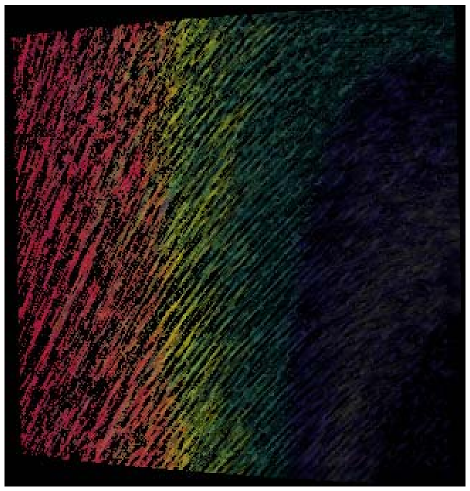
Figure 5.10 shows the results of applying the different distributions. It shows the effects on a particular plane in the volume.



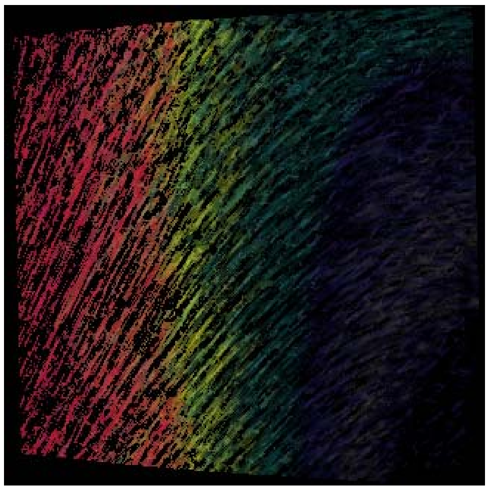
(a)



(b)



(c)



(d)

Figure 5.10: *Results of using different distributions*
(a) Poisson disk distribution (b) Jittering (c) Hammersley points (d) Halton points

The performance of these distributions in terms of the time taken to generate the points is discussed in the next chapter. In spite of the fact that introducing sparsity helps us to gain a global view of the vector field, we have to note that some details of the field might be missed as a result of creating empty space between the streamlines. Yet it aids the user in gaining a good understanding of the vector field.

5.2 Viewing the vector field locally

The previous section discussed ways to get a clear global view of the vector field. Introducing sparsity, transparency, encoding magnitude with color, highlighting areas with larger magnitude etc. help to get a better view of the field. The next section focuses on ways to view local, specific areas of the field and regions of interest. It discusses techniques such as using clipping planes and 3D patterns.

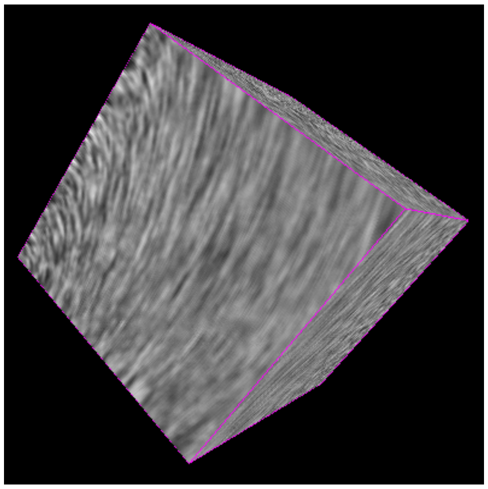
5.2.1 Cutting planes

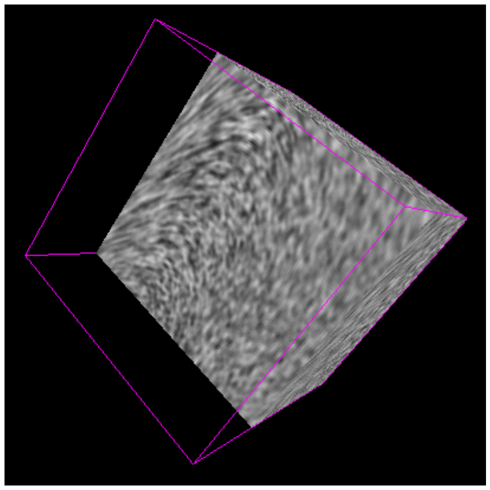
Salama et al. [2] have described and used different clipping mechanisms to view more localized regions in the vector field. We have made use of OpenGL clipping planes to allow the user to interactively clip the volume and view regions of interest on the inside.

The view volume that is defined forms six cutting planes. Objects that lie outside this volume are clipped and are not drawn in the final scene [24]. OpenGL allows us to define six additional clipping planes to remove unwanted parts of the object and thus to clip the volume further. Each of these planes is defined by the coefficients of the plane equation: $Ax + By + Cz + D = 0$. These clipping planes are transformed by the modeling and viewing transformation matrices. Thus the final volume that is displayed is the one that passes the clip tests of the view volume and user defined clipping planes.

5.2.1.1 Implementation

Clip planes are defined using the command *glClipPlane()*, which takes the plane number and the coefficients of the plane equation as its arguments. The corresponding clip plane is enabled using the command *glEnable()*. The vertices that emerge from the vertex processor are the ones that are clipped against the planes. Thus they should be in the same coordinate space as the planes [23]. The user defined clipping planes are defined in eye coordinates. Hence the vertices are converted to eye coordinates in the vertex shader and written to the output variable *gl_ClipVertex*. The results are undefined if the output variable is undefined. In our implementation, we have made use of sliders to allow the user to vary the value of D in the plane equation, thus allowing the user to move the clipping planes through the volume as shown in Figure 5.11.





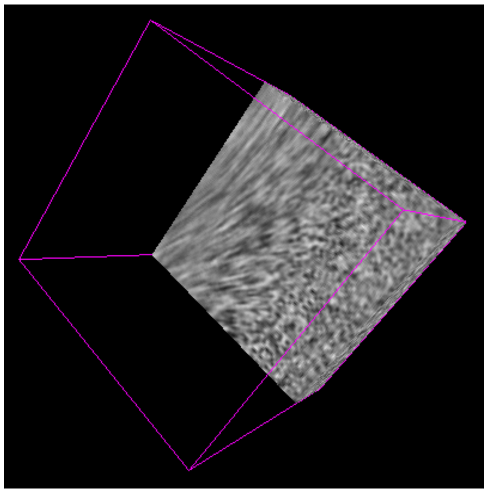


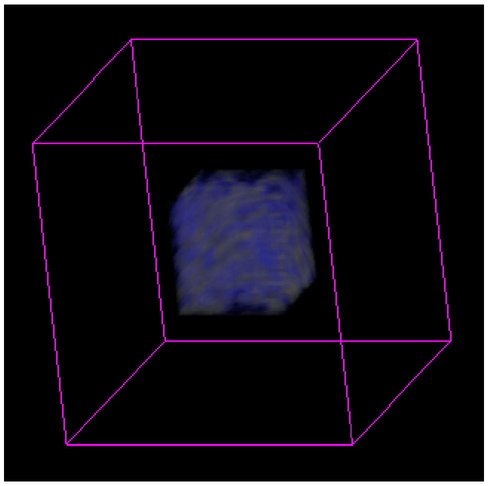
Figure 5.11: *Exploring the vector field interactively by moving the clip planes*

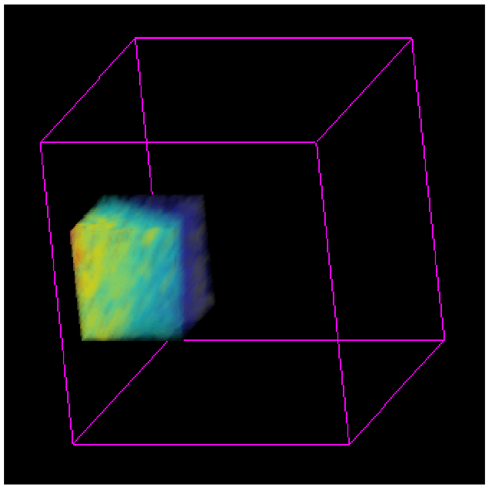
5.2.2 Regions of interest with 3D patterns

3D patterns allow the user to select a region of interest and explore it interactively. Our interface allows the user to choose one of several 3D patterns (such as cube, sphere, and cylinder) with an appropriate size for the pattern. The user can then translate the pattern to a desired region in the volume and explore it by using clip planes or by rotating the selected pattern.

5.2.2.1 Implementation

We have implemented patterns by defining a radius in the case of spheres and cylinders and by defining a width in the case of cubes. They are defined with respect to a center. In the fragment shader, all fragments that fall outside the radius/width are discarded so that only the part of the volume that falls within the 3D pattern is displayed. Thus the user is able to have a closer examination of a specific region of interest alone. Our interface provides sliders using which the user can change the center of the pattern, thus allowing the user to translate the pattern and slide it across the volume. When the pattern translates through the volume it has the appearance of moving through the volume. The user can also rotate the selected region alone. To get the effect of rotating the pattern through the vector field, the texture coordinates are also rotated by the same amount as the vertices. To do this rotation, the texture coordinates are translated to the origin, rotated and then translated back again. The texture coordinate transformations are saved in the texture matrix that is accessed in the vertex shader as *gl_TextureMatrix[i]*, where i is the texture unit number. The texture coordinates are multiplied by this matrix in the vertex shader to apply the texture transformations. The following figure shows the use of patterns to explore different parts of the vector field.





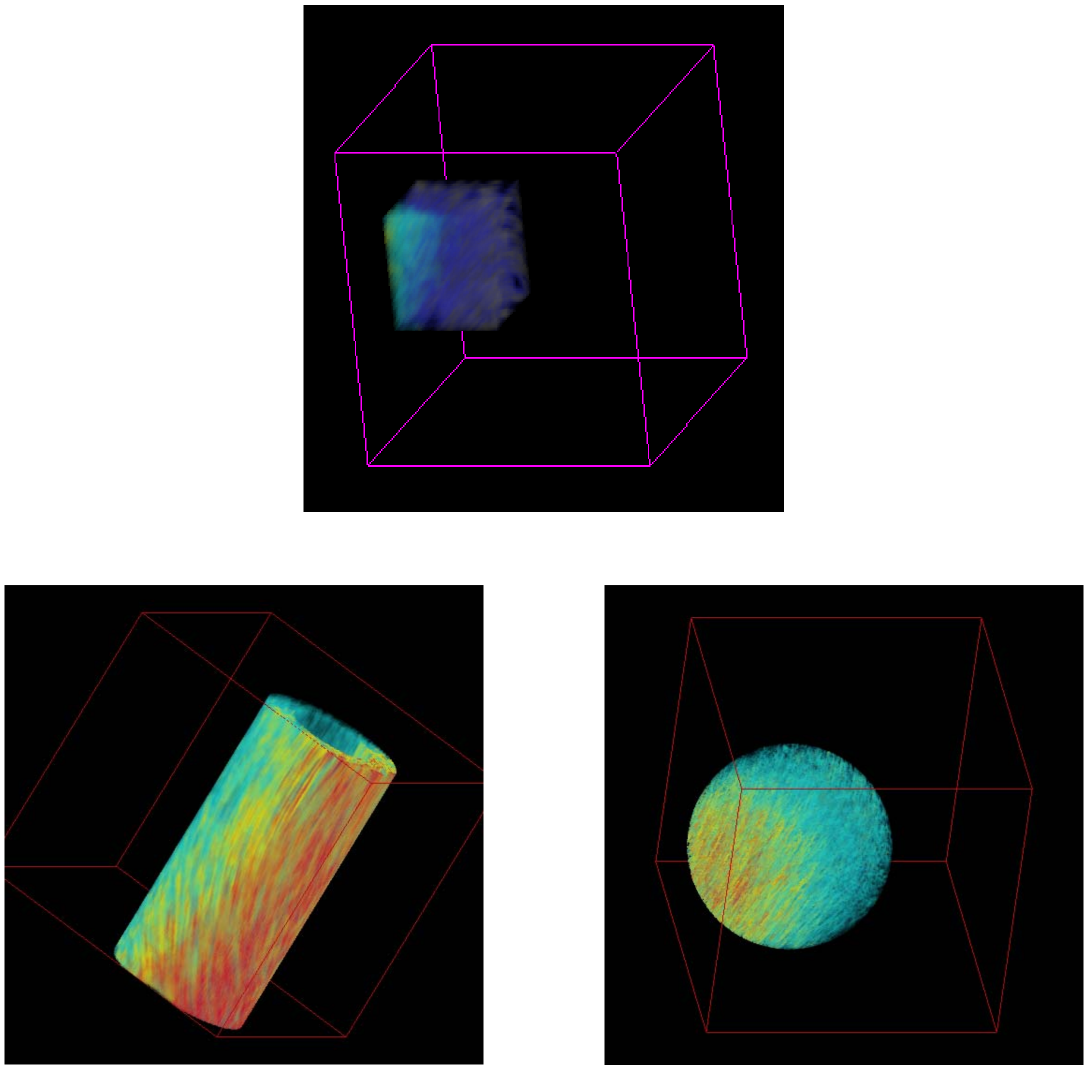


Figure 5.12: *Exploring the vector field using different patterns and translating/rotating them*

5.3 Improving Depth Perception

The previous section discussed ways of examining the local parts of the vector field. Next, we discuss another aspect of 3D field visualization. Spatial organization and depth perception play an important role in the effective visualization of 3D vector fields. This section discusses ways on improving the perception of the depth relation of the different parts of the field.

5.3.1 Intensity Depth cueing

Intensity depth cueing can be used to obtain rendering effects like fog, haze, etc. It can also be used to make objects far away from the user get darker with distance from the user. Hence depth perception is increased by having objects blend into the background color.

5.3.1.1 Implementation

We enable intensity depth cueing with the command *glEnable()* with GL_FOG as its argument. A fog blending factor f is used to blend the background color with the color of the incoming fragment. We use the linear mode of blending in which f is calculated using the following equation

$$f = \frac{(end - z)}{(end - start)}$$

where z is the eye-coordinate distance between the viewpoint and the fragment center

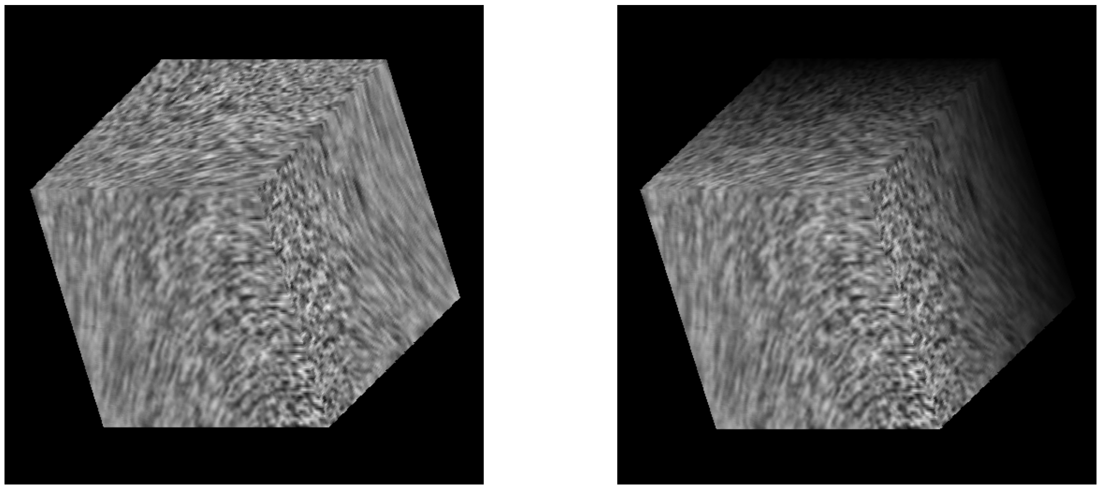
start is the distance to the start of the fog effect

end is the distance to the end of the effect [24].

The fog mode, the start value, end value and the background color are specified using the *glFog()* function.

In the vertex shader, the z value that is specified in the equation is calculated for each vertex in the eye coordinate space and is written to the special output variable *gl_FogFragCoord*. The fog blending factor f is then calculated in the fragment shader according to the above equation. Then the final color of the incoming fragment in the fragment shader is calculated as the linear blend of the original color and the background color. This is done using the GLSL *mix()* function. Our interface allows the user to vary the start and the end values using sliders.

Figure 5.13 (b) shows fog effects. The fog color is taken to be black. The part of the volume far away from the user has more fog applied to it than the part of the volume nearer to the user. This method is useful for recognizing far away objects from nearby ones. However, it is not efficient in portraying the minute depth differences.



(a) *(b)*

Figure 5.13: *Fog (a) Without Fog (b) With Fog*

5.3.2 Stereographics

In this section, we discuss the use of stereo graphics to de-clutter a complex 3D scene [30]. When we look at the world around us, 3D stereographic effects arise from the fact that our left and right eyes see the world from slightly different perspectives. Perception of depth is improved greatly by using stereo graphics which uses both the left and the right eye views.

If the left and the right eye views are generated in the orthographic projection mode, then perspective shortening causes a point to have different vertical positions in both the views. This is known as vertical parallax. To avoid it, stereographics is generally done in the perspective mode. The left eye view is obtained by translating the eye by –E in the X direction. This is done by translating the scene by +E in the X direction. The right eye view is obtained by translating the scene by –E in the X direction. Here (2*E) is the horizontal distance between the left and the right eye views.

The function *glFrustrum()* is used to specify a monoscopic perspective projection viewing frustum. This function takes the left, right, bottom, top, near and far planes of the near clipping plane as its arguments. We define the plane of zero parallax as the plane where a 3D point projects to the same window location for both eyes. The left (L0p), right (R0p), bottom (B0p) and top (T0p) boundaries of the viewing window on the plane of zero parallax are measured as:

L0p = -Z0p*tan($\phi / 2$)

R0p = +Z0p*tan($\phi / 2$)

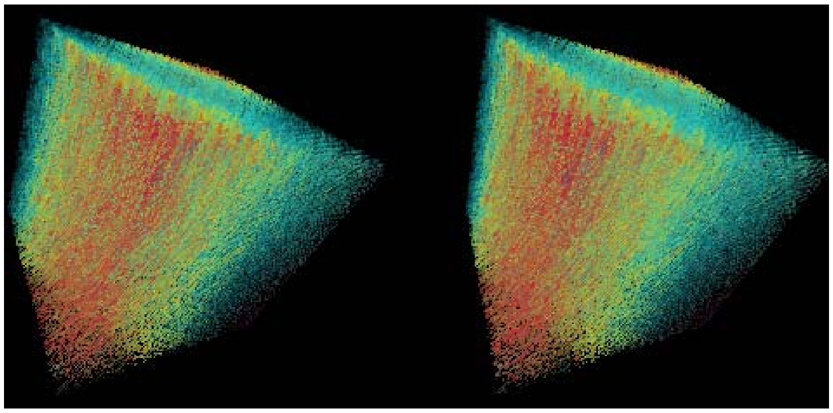
B0p = -Z0p*tan($\phi / 2$)

T0p = +Z0p*tan($\phi / 2$)

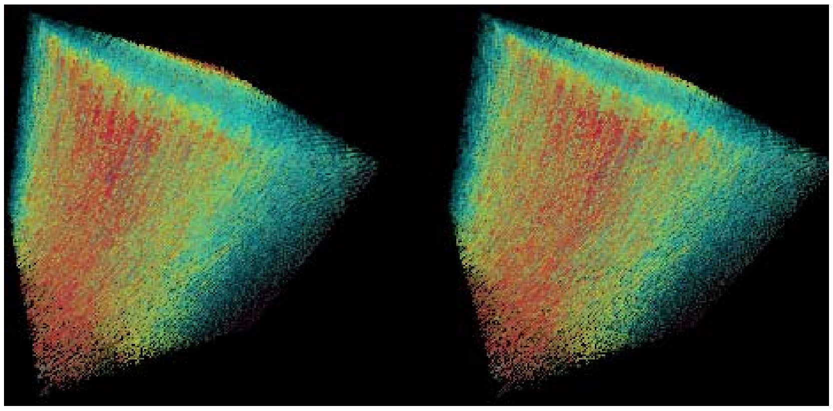
where Z0p is the distance to the plane of zero parallax and ϕ is the field-of-view angle. Next the boundaries in the left eye view are shifted by +E to match the +E shift in the scene. Similarly, the boundaries in the right eye view are shifted by –E. Only the left and the right boundaries are affected by this scene shift.

5.3.2.1 Implementation

We enable stereo graphics by or'ing GLUT_STEREO into the argument of the function *glutInitDisplayMode()*. Then we draw the left eye view into the left back buffer and the right eye view into right back buffer. Using appropriate glasses, the left and the right eye views (as shown in Figure 5.14) can be combined to give an image with better depth order relationships. We find stereographics to be a very effective means of providing depth perception.



(a)



(b)

Figure 5.14: *Using stereo graphics to de-clutter the field*
(a) Left – Right View (b) Right – Left View

5.3.3 Lighting

We now discuss how applying lighting helps to improve the overall visualization of the 3D vector field. Lighting helps to brighten up areas of the image which otherwise appear dull. Hence certain parts of the image appear bright and are highlighted. Specular highlights help to establish a spatial relationship with respect to the position of the light source.

5.3.3.1. Implementation

Lighting is applied by the following procedure:

- A normal is assigned to each point. The normals can be passed to the shaders using the command *glNormal()* in OpenGL. However, we use the normals only when we need to apply lighting. Hence we recomputed the normals and pass the normals as a texture to the fragment shader.
- The normal for the incoming fragment is accessed in the fragment shader. Then the color as the result of applying lighting is calculated by the following equation:

  $C_l = C_a + C_d * \max(\mathrm{abs}(\mathrm{dot}(N,L)),0.0)) + C_s * \mathrm{pow}(\max(\mathrm{dot}(R,E),0.0),C_{oeffs})$

  where C_l is the color obtained after lighting calculations

  C_a is the ambient color of light

  C_d is the diffuse color of light

  C_s is the specular color of light

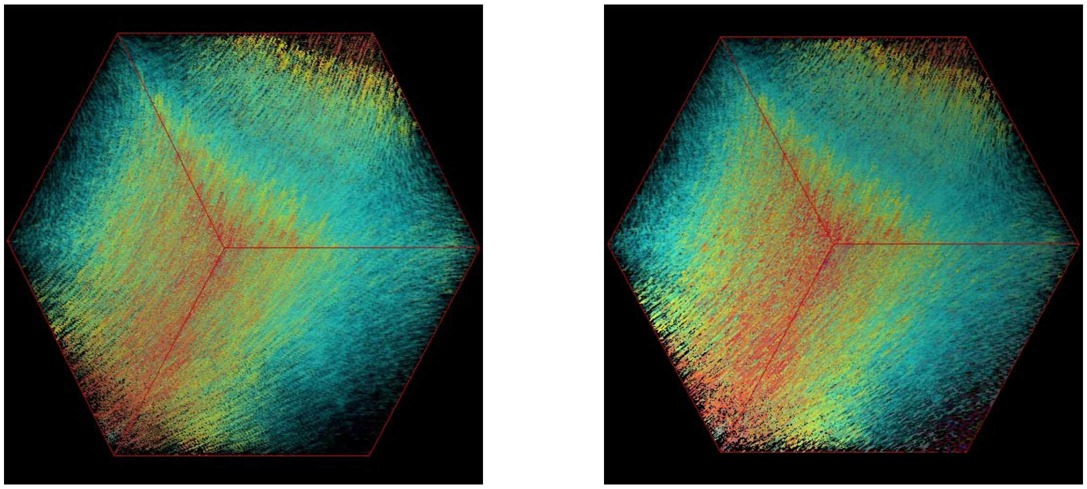
  C_{oeffs} is the specular coefficient

  N is the normal at that position. It is obtained by considering the vector field at that point as the tangent. The normal is thus calculated considering the local TNB coordinate system.

  L is the light vector that is calculated as the vector Light Position - Vertex Position

R is the reflection vector that is calculated using the *reflect()* function
E is the eye vector that is calculated as the vector Eye Position - Vertex Position

The final color is obtained by blending the color obtained after applying LIC with Cl. The Figure 5.15 (b) shows the results of applying lighting.



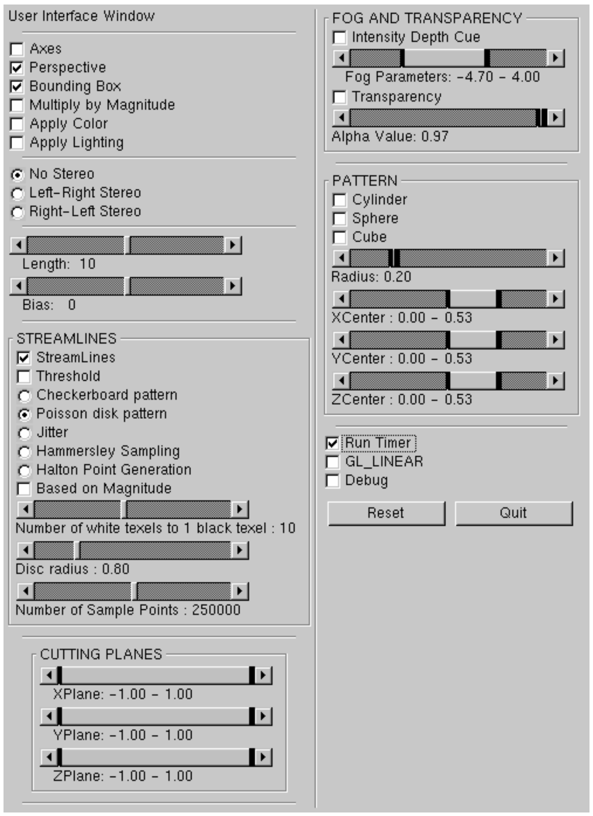
(a) *(b)*

Figure 5.15: *Lighting (a) Without Lighting (b) With Lighting (depth differences are seen)*

6. RESULTS

In this chapter, we discuss the results that were obtained. We have made use of the graphics processing unit (GPU) to accelerate the computation of LIC in order to visualize 3D vector fields at interactive rates. The implementation is done in C++ and OpenGL. GPU programming is done in the OpenGL Shading Language (GLSL) [23]. The entire work is done on a 4 GHz Pentium 4 CPU with 2GB of RAM, with an NVIDIA Quadro 3400 FX graphics card with 256MB of memory.

Figure 6.1 shows the user interface window, created using the GL User Interface toolkit (GLUI). Range sliders (a GLUI extension our research group added) are provided to let the user vary range parameters like the length and bias. The user also has the option of viewing the vector field with or without sparsity and of choosing the distribution to be used to produce the sparse input texture. The user can vary the ratio of the white texels to black texels in the case of checkerboard distribution, the disc radius in the case of jittering and the number of sample points in the case of Hammersley and Halton distribution. Range sliders are also provided for the clipping planes along the x, y and z axis. The fog parameters and the alpha value can also be changed using the sliders. The radius and the center of the 3D patterns used to view localized regions of interest can be changed by the user as well. The user can interactively change the length and bias values, control sparsity, use the clip planes to view local planes, introduce transparency and view localized regions of interest. Thus the user can interactively explore the entire 3D volume.



User Interface Window

[ ] Axes
[✓] Perspective
[✓] Bounding Box
[ ] Multiply by Magnitude
[ ] Apply Color
[ ] Apply Lighting

[✓] No Stereo
[ ] Left–Right Stereo
[ ] Right–Left Stereo

Length: 10
Bias: 0

STREAMLINES
[✓] StreamLines
[ ] Threshold
[ ] Checkerboard pattern
[✓] Poisson disk pattern
[ ] Jitter
[ ] Hammersley Sampling
[ ] Halton Point Generation
[ ] Based on Magnitude
Number of white texels to 1 black texel : 10
Disc radius : 0.80
Number of Sample Points : 250000

CUTTING PLANES
XPlane: –1.00 – 1.00
YPlane: –1.00 – 1.00
ZPlane: –1.00 – 1.00

FOG AND TRANSPARENCY
[ ] Intensity Depth Cue
Fog Parameters: –4.70 – 4.00
[ ] Transparency
Alpha Value: 0.97

PATTERN
[ ] Cylinder
[ ] Sphere
[ ] Cube
Radius: 0.20
XCenter : 0.00 – 0.53
YCenter : 0.00 – 0.53
ZCenter : 0.00 – 0.53

[✓] Run Timer
[ ] GL_LINEAR
[ ] Debug

Figure 6.1: *User Interface*

6.1 Timing Benchmarks

6.1.1 If Constructs

We have investigated the timing of some GLSL constructs. The presence of *if* statements has a large effect on the performance of the fragment shader. This makes sense, since the GPU is essentially a SIMD computing device. The table and graph show the performance of the fragment shader with and without the *if* statements in a *for* loop.

Table 6.1: *Effect of If constructs*

LIC Length	Without *if* (in FPS)	With *if* (in FPS)
2	10 -12	7.5 - 9.5
4	6 - 7	4.5 - 6
6	4.5 - 6	3.5 - 4.5
8	3.5 - 4.5	2.5 - 3
10	2.7 - 4	2 - 2.6

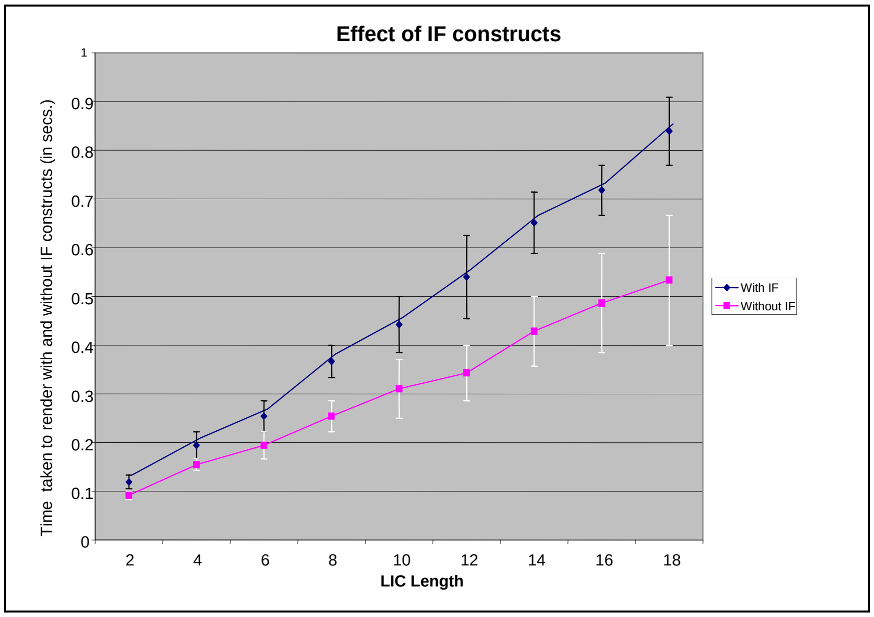


Figure 6.2: *Performance of the GPU with and without If constructs*

But, with all the different display modes, we really need the *if* tests in the fragment shader. One approach would be to have different versions of the fragment shader, and load the appropriate one every time the user changes display modes. But, maintaining these many versions of the fragment shader would raise the ugly possibility of version skew. In our implementation, we have used the preprocessor directive *#ifdef* to simulate *if* blocks. We load four “different” fragment shaders. The fragment shaders differ only in the values of the *#define* statements pre-appended to them. The first time an option is selected, the proper *#ifdef* is set and the fragment shader is linked and compiled. When the option is selected again, the corresponding code is just executed. Hence the efficiency of the code is improved without maintaining multiple versions of it. This is found to give much better performance than using the *if* statements or by loading the appropriate shader every time the user switches display modes.

6.1.2 Resolution and Length

As the LIC length increases, the frame rate drops as more texels have to be accessed in the fragment shader, on the local streamline at every point. Also, when the resolution is less, the frame rate is more. We have tested with a 128*128*128 and a 64*64*64 volume data. The performance graph is shown in Figure 6.3. The rendering time differs approximately by a factor of 2, since for a 128*128*128 volume, we draw twice as many planes in each direction. However, each plane takes the same time since it accesses the same number of fragments.

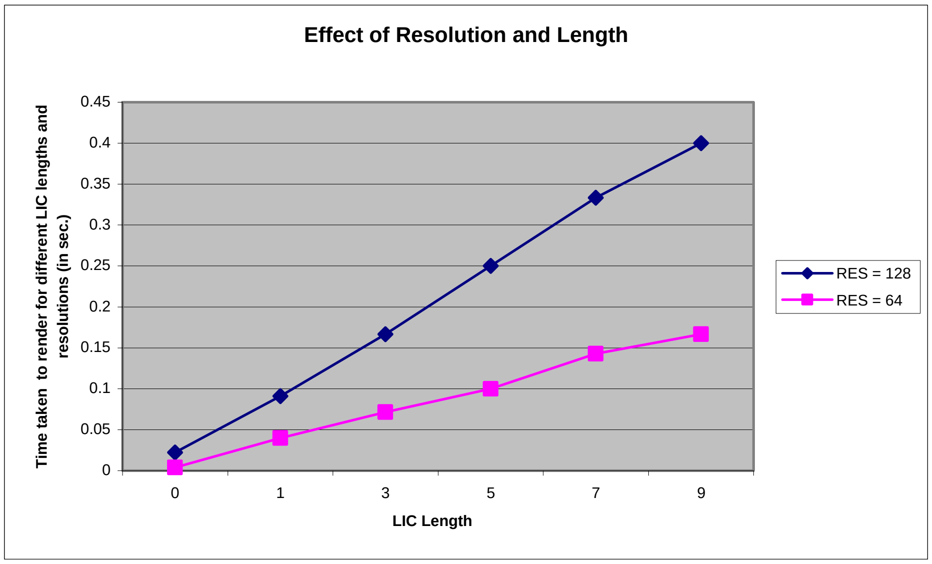
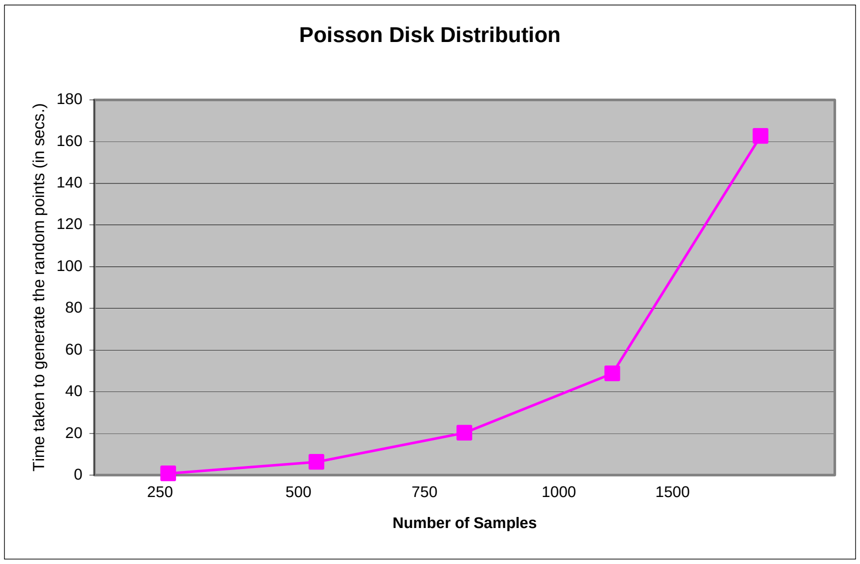


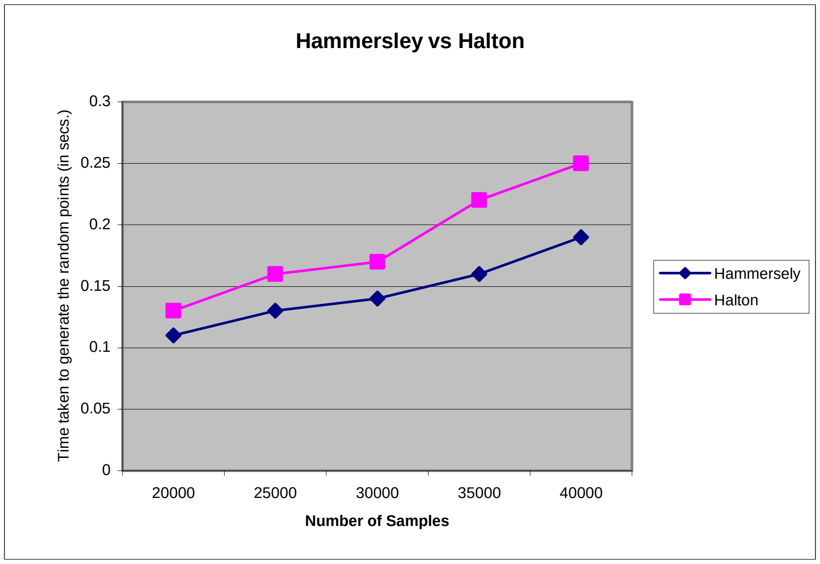
Figure 6.3: *Effect of resolution and length on performance*

6.1.3 Different distributions

We have introduced sparsity as a means of de-cluttering the otherwise dense 3D scene, obtained as a result of applying LIC to the noise texture. In chapter 5, we discussed the various techniques that we used to obtain the sparse input texture. To recap, we have used the Poisson-Disk distribution, jittering, Halton and Hammersley distributions to generate a set of evenly distributed points in the sparse input texture. As seen in Figure 6.4(a), Poisson disk is an expensive method. Particularly, as the number of points increase, the time taken to generate them increases to a great extent. Also, as shown in Figure 6.4(b), when compared to Halton point generation, the Hammersley method is slightly faster. This is because in this method, one reflection about the prime base is substituted by a division operation, which is faster. Jittering always takes the same time since irrespective of the number of samples found; each and every point needs to be checked against its neighbors. Hence in jittering, the time complexity is independent of the number of samples needed.



(a)



(b)

Figure 6.4: *Performance of the distributions (a) Performance of Poisson disk distribution (b) Performance of Hammersley point generation vs. Halton point generation*

6.1.4 Effect of size on the screen

As the size of the volume increases on the screen, the frame rate drops. This is because it takes longer to render the volume on the screen. This is shown in the graph in Figure 6.5. We have tested with different sizes of the volume on the screen and for different lengths of LIC. Irrespective of the length, the time taken to render the volume is lesser when it size is lesser. Again, for different lengths, the time taken to render the volume for smaller lengths of LIC is lesser.

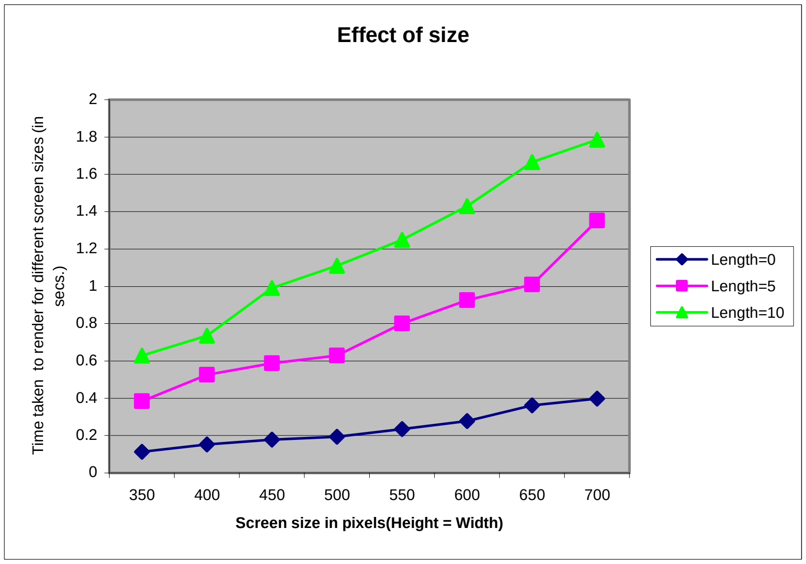


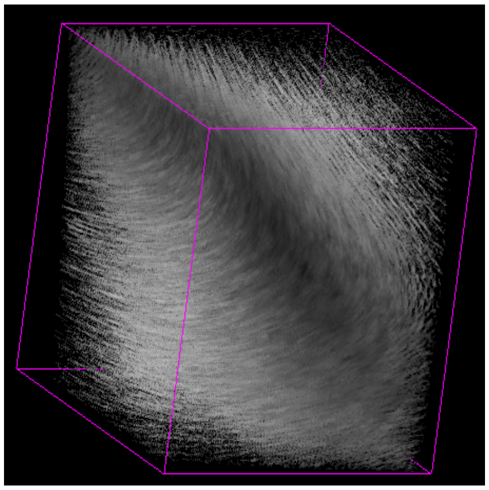
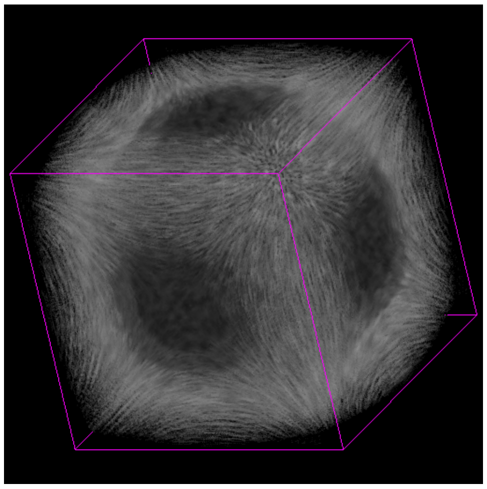
Figure 6.5: *Effect of size on the screen*

6.2 Vector Fields Used

We have tested our implementation of LIC using two different vector fields. The vector field shown in Figure is created using an equation that describes flow around a corner [19]. The field in Figure 6.6(b) is created using an equation that describes a solenoidal vector field. A solenoidal vector field has zero divergence. It is generated using the equation:

$$V_x = (y^2+z^2)yz$$
$$V_y = (x^2+z^2)xz$$
$$V_z = (x^2+y^2)xy$$

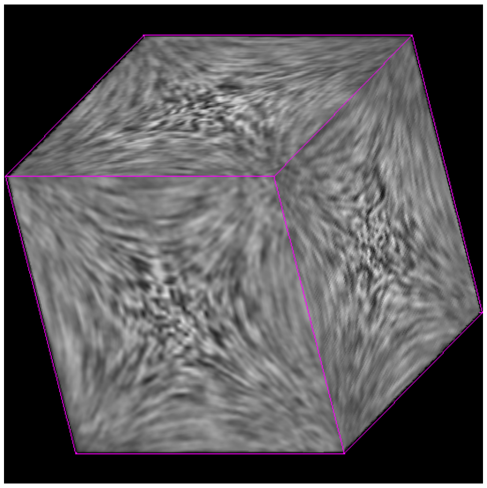
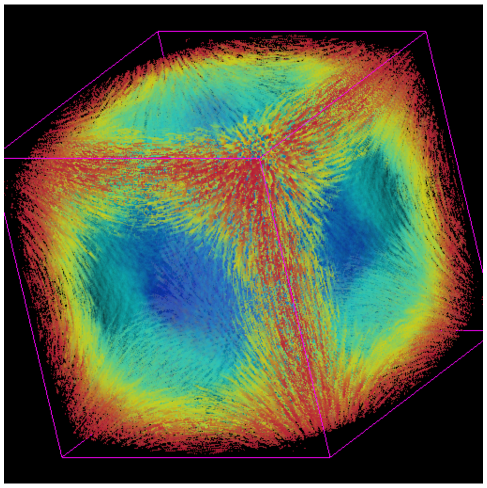
where V_x, $V_{y,}$ and V_z are the x, y and the z components of the vector field [27].

(a) *(b)*

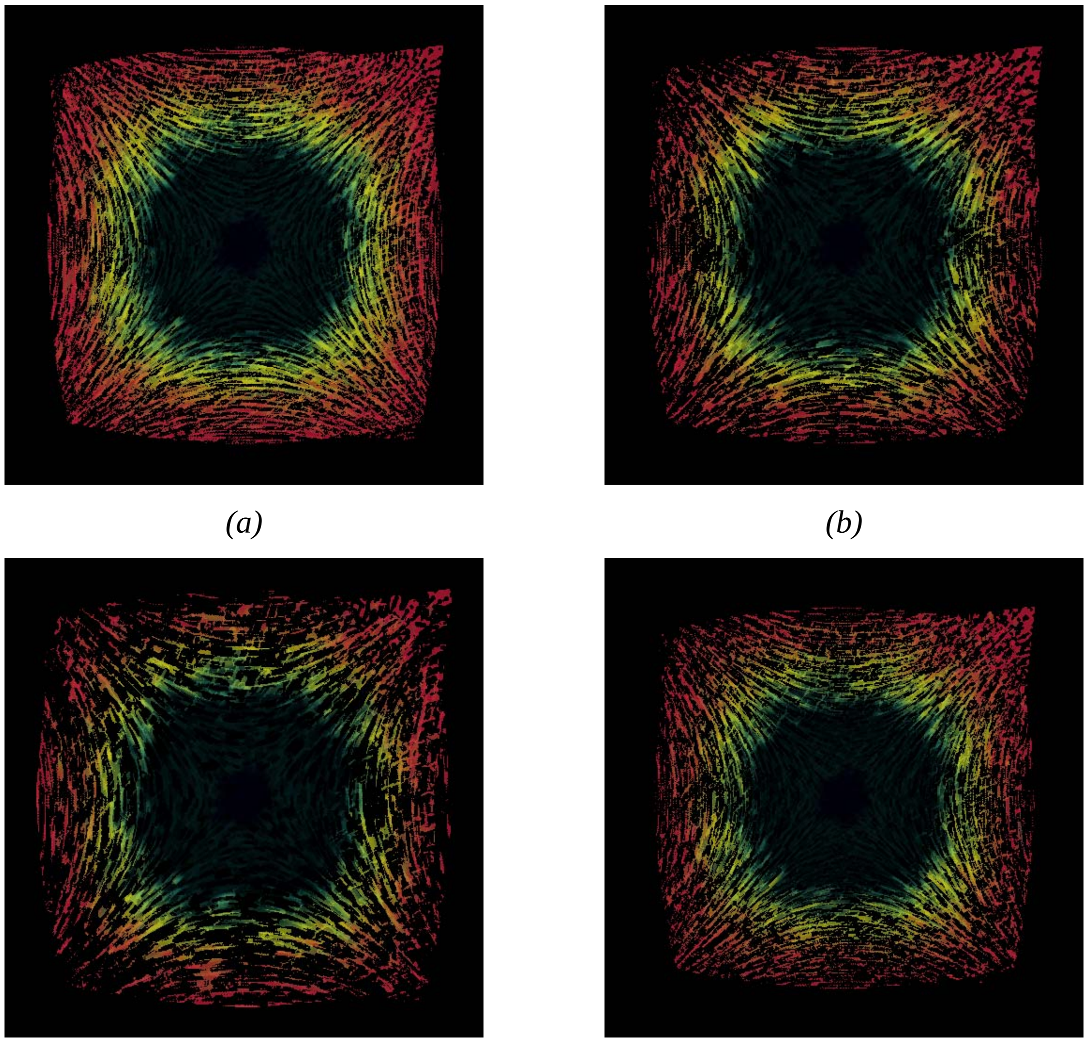
Figure 6.6: *Different vector fields*
(a) Flow around a corner (b) A solenoidal vector field

The following images show the results that we obtained with the solenoidal vector field.

(a) *(b)*

Figure 6.7: *Solenoidal field (a) Solenoidal vector field (b) With sparsity (Hammersley distribution)*



(a) (b) (c) (d)

Figure 6.8: *Sparsity with different distributions (a) Poisson disk distribution (b) Halton points (c) Jittering (d) Hammersley points*

6.3 Zoom In

The purpose of introducing sparsity is that the observer can peer through the stream lines to get a better look of the inner part of the vector field and thus understand the inner details of the field. This is shown in Figure 6.9. Note that Figure 6.9 looks very much like zooming into an Impressionist painting. We believe that there are some interesting interactive painterly rendering opportunities with this method.

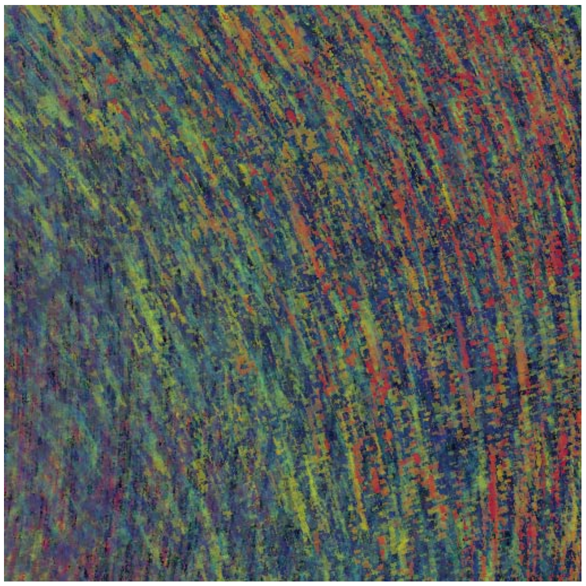
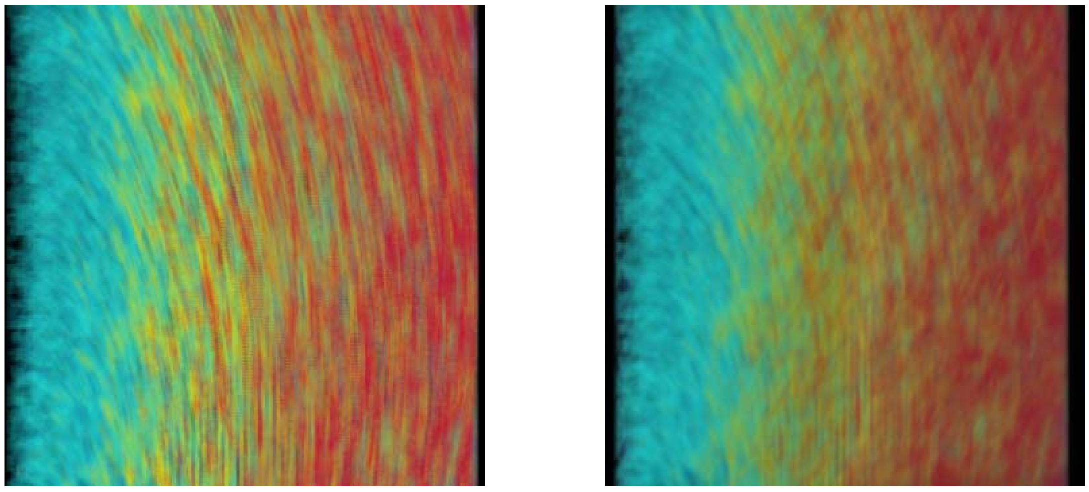


Figure 6.9: *A zoomed in view*

6.4 Nearest Vs. Linear Filtering

Nearest filtering takes into account the color of the texel in the texture map that is closest to the center of the point being textured. Linear filtering takes the weighted average of the colors of the eight neighboring texels (in the case of 3D texture maps). Nearest filtering is slightly faster, but linear filtering produces nice smooth lines. In our case however, there is more value in clearly viewing the streamlines than in having a smooth blended appearance. Nearest filtering (Figure 6.10(a)) does a better job of maintaining the distinction between individual streamlines than linear filtering (Figure 6.10(b)) which blurs the lines too much. Hence we make use of nearest filtering for our application.



(a) *(b)*

Figure 6.10: *Effects of different types of filtering for texture mapping (a) Nearest filtering (b) Linear filtering*

6.5 Limitations of 3D LIC and how they are overcome

The following are some of the disadvantages of using LIC for 3D vector field visualization. We follow each limitation with a note on how we overcome such limitations.

(a). The computation of streamlines is expensive. The cost of computation increases as the length of the convolution kernel (i.e. the user defined length) increases. This is because at each point, more pixels have to be accessed. *The use of GPU speeds up this process.*

(b). As shown in the images of Figure 4.4, LIC provides only a global behavior of the 3D vector field. The resulting images are cluttered and limited information is available about the local nature of the field. *Cutting planes and 3D patterns enable the user to explore local regions.*

(c). The resulting images are dense. Hence it is difficult to perceive the flow. The flow on the inside of the volume is not seen. *Introduction of sparsity overcomes this limitation.*

(d). As seen in Figure 6.11, unnecessary details (such as regions of low magnitude which the user is not interested in viewing) are present in the flow. In Figure 6.11 the highlighted areas indicate areas of the vector field where the magnitude of the field is very low. They represent areas of the vector field that are of little or no significance. In some applications it may be desirable that they are not present in the output image. *Varying opacity as a function of magnitude highlights regions of higher magnitude.*

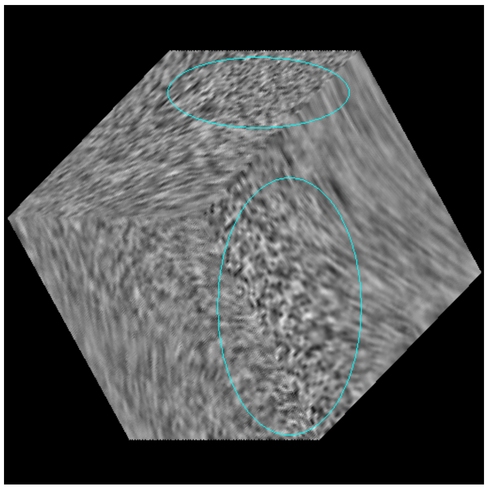


Figure 6.11: *Unnecessary details in the 3D LIC image*

(e) Scalar information of the flow like the magnitude of the vector field is not present. *Color coding of the vector field incorporates such scalar information.*

(f) Spatial organization of the vector field is not clear. Subtle depth differences are not perceived. i.e. the relative positions of the streamlines are not portrayed effectively. *The use of stereographics and lighting improves depth perception.*

7. CONCLUSIONS AND FUTURE WORK

In this project we have implemented several techniques to allow the users to gain a good insight into the three-dimensional vector field under study that is displayed using the LIC algorithm. These techniques make use of the capabilities of the GPU to speed up the process. The methods that we have employed, such as the introduction of sparsity and the use of stereographics, make it easier for the user to explore and analyze the 3D vector field, which is otherwise dense and cluttered. Overall, we are very pleased with using the GPU to perform interactive 3D LIC analysis. The speed allows us to change LIC parameters, employ the various techniques and have the display respond at interactive rates. This has given us a much more complete insight into the overall nature of a 3D flow field, unlike anything we have achieved with other methods.

One interesting characteristic of fragment program methods is that they are generally window-size dependent, not data-size dependent. That is, increasing the size of the window causes more fragments to need to be drawn, which increases the number of calls to the fragment code. This characteristic cuts both ways. The good news is that the viewer can automatically increase the density of the LIC display simply by enlarging the window. The bad news is that the display time goes up with the square of window dimension.

However, GPU speed trends soften that bad news. As shown in Figure 7.1, GPU performance is on an even steeper path than Moore’s-law general processors. Thus, we know that any visualization methods that employ GPU programming are going to be faster in the future. This will let us deal with larger 3D flow datasets and higher resolution displays.

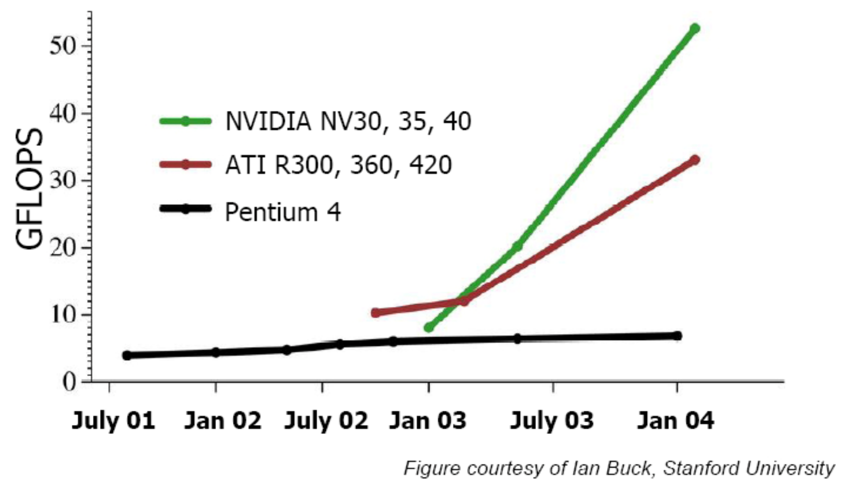


Figure 7.1: *GPU Speed Trends*

Also, one of the disadvantages of using shaders is that debugging is difficult. There is no standard way to help programmers debug and detect errors.

LIC volumes are inherently difficult to look at in that the data in front hides the data behind it. Among the methods that we used to introduce sparsity, Poisson-Disk distribution gives good results. Changing the number of points interactively was not possible due to the complexity of the algorithm. One venue for future work would be to use more complex data structures and implement quicker versions of Poisson-Disk distribution.

Further, we would like to expand our techniques to incorporate the capability to visualize unsteady vector fields as well. We would also like to analyze how the increase in the number of vertex and fragment processors on the GPU will increase the speed of our LIC implementation.

REFERENCES

[1] Brian Cabral, Leith (Casey) Leedom. Imaging vector fields using line integral convolution. *Proceedings of SIGGRAPH '93*, pages 263-270.

[2] Rezk-Salama C., Hastreiter P., Teitzel C., Ertl T. Interactive exploration of volume line integral convolution based on 3D-texture mapping. *Proceedings IEEE Visualization' 99*, pages 233--240.

[3] Han-Wei Shen, Christopher R. Johnson, Kwan-Liu Ma. Visualizing vector fields using line integral convolution and dye advection. *Symposium on Volume Visualization*, pages 63--70, 1996.

[4] Detlev Stalling, Hans-Christian Hege. Fast and resolution independent line integral convolution. *Proceedings of SIGGRAPH '95*, pages 249-256.

[5] Victoria Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. *Proceedings of SIGGRAPH '97*, pages 109 – 116.

[6] Victoria Interrante, Chester Grosch. Strategies for effectively visualizing 3D flow with volume LIC. In *Yagel and Hagen*, editors, *Proceedings of Visualization '97*, pages 421-424.

[7] J.J. van Wijk. Spot noise-texture synthesis for data visualization. *Proceedings of SIGGRAPH '91*, volume 25, pages 263--272.

[8] Jarke J. van Wijk. Image-based flow visualization. *Proceedings of SIGGRAPH'2002*, pages 745-754.

[9] Heidrich, W., Westermann, R., Seidel, H.-P., Ertl T. Applications of pixel textures in visualization and realistic image synthesis. *ACM Symp. Interactive 3D Graphics*, pages 127 – 134, 1999.

[10] Bruno Jobard, Gordon Erlebacher, M.Yousuff Hussaini. Hardware accelerated texture advection for unsteady flow visualization. *Proceedings of the IEEE Visualization' 2000*, pages 155 – 162.

[11] Oleg A.Potiy, Alexey A.Anikanov. GPU-based texture flow visualization. *Proceedings of Graphicon,* International Conference on Computer Graphics and Vision, 2004.

[12] Daniel Weiskopf , Thomas Ertl. GPU-based 3D texture advection for the visualization of unsteady flow fields. *Proceedings of WSCG'2004*.

[13] Robert L.Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, vol 5, issue 1, pages 51-72, 1986.

[14] Don P.Mitchell. Spectrally optimal sampling for Distribution Ray Tracing. *Proceedings of SIGGRAPH' 91*, pages 157—164.

[15] Tien-Tsin Wong, Wai-Shing Luk, Phen-Ann Heng. Sampling with Hammersley and Halton points. *Journal of Graphics Tools* , vol. 2, no. 2, 1997, pages 9-24.

[16] Malte Zockler, Detlev Stalling, and Hans-Christian Hege. Interactive visualization of 3D-vector fields using illuminated streamlines. *Proceedings of IEEE Visualization '96*, pages 107-- 113.

[17] Xiaoqiang Zheng, Alex Pang. HyperLIC. *Proceedings of IEEE Visualization 03*, pages 249-256.

[18] http://www.princeton.edu/~asmits/Bicycle_web/streamline.html

[19] David Knight and Gordon Mallinson. Visualizing unstructured flow data using dual stream functions. *Visualization and Computer Graphics*, IEEE Computer Society, Vol 2, No 4, pages 355 – 363, 1996.

[20] *Interactive Computer Graphics, A Top-Down Approach using OpenGL.* Edward Angel. Third edition, Addison Wesley, 2002.

[21] *Fundamentals of Computer Graphics*. Peter Shirley. Second edition, AK Peters Ltd., 2005.

[22] *The Visualization Handbook*. Charles D.Hansen, Christopher R.Johnson. Academic Press, 2004.

[23] *OpenGL Shading Language*. Randi Rost. Addison-Wesley, 2006.

[24] *OpenGL Programming Guide*. Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis. Fifth edition, Addison Wesley, 2005.

[25] http://www.cs.unc.edu/~rademach/glui/src/release/glui_manual_v2_beta.pdf

[26] *Let us C*. Yashavant Kanetkar. Third edition, BPB publications, 1999.

[27] *Vector Analysis for Engineers and Scientists*. P.E.Lewis and J.P.Ward. Addison Wesley, 1989.

[28] http://www.lighthouse3d.com/opengl/gls

[29] http://web.cs.wpi.edu/~matt/courses/cs563/talks/antialiasing/stochas.html

[30] http://web.engr.oregonstate.edu/~mjb/cs553/Handouts/Stereo/stereo.pdf

[31] http://www.bluevoid.com/opengl/sig99/advanced99/notes/node314.html

[32] http://developer.3dlabs.com/documents/

[33] http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf

[34] http://mew.cx/glsl_quickref.pdf