View-Dependent Multi-resolutional Flow Texture Advection

Liya Li[a] and Han-Wei Shen[a]

[a] Department of Computer Science and Engineering, The Ohio State University, Columbus, Ohio

ABSTRACT

Existing texture advection techniques will produce unsatisfactory rendering results when there is a discrepancy between the resolution of the flow field and that of the output image. This is because many existing texture advection techniques such as Line Integral Convolution (LIC) are inherently none view-dependent, that is, the resolution of the output textures depends only on the resolution of the input field, but not the resolution of the output image. When the resolution of the flow field after projection is much higher than the screen resolution, aliasing will happen unless the flow textures are appropriately filtered through some expensive post processing. On the other hand, When the resolution of the flow field is much lower than the screen resolution, a blocky or blurred appearance will present in the rendering because the flow texture does not have enough samples. In this paper we present a view-dependent multiresolutional flow texture advection method for structured recti- and curvi-linear meshes. Our algorithm is based on a novel intermediate representation of the flow field, called trace slice, which allows us to compute the flow texture at a desired resolution interactively based on the run-time viewing parameters. As the user zooms in and out of the field, the resolution of the resulting flow texture will adapt automatically so that enough flow details will be presented while aliasing is avoided. Our implementation utilizes mipmapping and programmable GPUs available on modern programmable graphics hardware.

Keywords: flow visualization, texture advection, trace slices, multi-resolution techniques, GPU

1. INTRODUCTION

Effective visualization of flow fields plays an important role in understanding data generated from a variety of scientific applications such as computational fluid dynamics, climate modelling, and computational physics. Besides using the more traditional techniques such as arrow plots or particle tracing, texture advection has been accepted as an effective way to display simultaneously the flow's directions everywhere in the field. In general, existing texture advection techniques for structured recti- and curvi-linear grid data can be classified into object space and image space methods. In the object space methods such as,[1,2] the computation of textures is first performed in the domain that defines the flow field using techniques such as Line Integral Convolution (LIC). The resulting texture is then mapped to the proxy geometry representing the underlying surface and displayed on the screen. The image space methods such as IBFVS,[3] or ISA,[4] on the other hand, perform the calculation directly on the screen. In those methods, the computation is done at per fragment level through successive advection and blending of textures.

In general, the object space methods do not consider the viewing parameters related to the display when the texture advection is performed. As the resulting flow texture is mapped to the surface mesh, aliasing or distortion can happen if there is a large discrepancy between the density of the mesh and the resolution of the screen. When multiple grid cells are projected to a single pixel, for example, a direct mapping of the flow texture to the geometry will produce an aliasing result because the texture is under-sampled. When the density of the grid after projection is much lower than the screen resolution, on the other hand, the output texture does not possess enough granularity to depict the flow directions clearly because the texels in the flow texture will only get enlarged or interpolated. In fact, both aliasing and a lack of clear depiction of flow directions can exist simultaneously when the grid density varies substantially across the field domain. This is particularly common for data defined on curvilinear grids.

For the image space methods, when the underlying flow field is defined on a surface mesh existing in 3D space, the mesh is first projected onto the image plane before the texture advection and blending are performed. IBFVS[3] does the projection after the vertices are advected in the flow field while the ISA[4] algorithm projects both the vertices and the

Further author information:
Liya Li E-mail: lil@cse.ohio-state.edu
Han-Wei Shen E-mail: hwshen@cse.ohio-state.edu

vector field before the advection is performed. In those methods, since the input texture is defined and advected in the screen space, the texture aliasing problem is alleviated. Performing the texture advection after projection, however, may encounter several problems. First, since the input noise is defined in the screen space and thus has the same size and frequency everywhere regardless of the distances, forms, and sizes of the 3D objects, important cues for depth and shape reasoning in 3D space are lost. Also, when multiple cells are projected to the same pixel, since the advection and blending are performed in the image space, the path of the texture advection can be incorrect. Finally, the restriction of only using image space input textures makes it more difficult to control the appearance of the output, especially for the case when it is more desirable to advect textures that are adhering to the object surfaces in 3D space.

To address the issues described above, in this paper, we present a view-dependent flow texture advection algorithm based on a hybrid image and object space approach. The algorithm can be applied to 2D steady and time-varying vector fields defined on structured rectilinear and curvilinear surface meshes. Our algorithm is an image space method in the sense that the flow texture is computed for each fragment at the rasterization stage where the mesh has already been projected to the screen. It is an object space method because the input texture to be advected and the flow advection paths all exist in the original domain where the flow field is defined. Our goal is to preserve important depth cues that allow better depictions of 3D shapes. Our algorithm can generate flow patterns with sufficient granularity in the output image even at the places where the mesh is sparse. As the user zooms in and out of the field, the resulting flow texture of an appropriate resolution will be computed at an interactive speed. When the view is moving, our algorithm will not produce a blurred result as in those image space methods[3,4] where the texture will only clear up over the course of several frames.

The paper is organized as follows. We first review some existing flow texture advection techniques in section 2. We then describe our algorithm in section 3 and section 4, which include a novel representation of the flow field called trace slices, a texture advection algorithm, and an automatic level of detail adjustment mechanism. We present experimental results and discuss some characteristics of our algorithm in section 5, and conclude this paper with future work in section 6.

2. RELATED WORK

Texture advection has been widely used for visualizing flow fields. Compared to the more traditional particle advection methods, texture advection techniques can better show the global characteristics of the flow field without the need for sophisticated seed placement schemes[5,6] for streamline and streakline computation. Among the existing texture advection methods, Line Integral Convolution (LIC)[7] and Spot Noises[8] are generally considered classic. In LIC, convolution is performed along the streamline paths originated from each pixel in a 2D grid to create coherent flow patterns. In Spot Noise, random spots are warped along the local flow directions and blended together to create the final image. Both of the algorithms have inspired a substantial amount of follow-up research in the past decade.[9–13]

Both LIC and Spot Noise are primarily software based methods. As graphics accelerators become faster and widely available in consumer level personal computers, there is a trend to move the texture advection operation to hardware. van Wijk proposed an Image Based Flow Visualization algorithm (IBFV),[14] which advects the underlying 2D mesh by the flow field. Through successive updates of texture coordinates at the mesh vertices, an input texture is being continuously advected and blended. Jobard *et al.*proposed a Lagrangian-Eulerian Advection (LEA) algorithm[15] for unsteady flows which performs the texture advection at each fragment at an interactive speed. For non-parametric surfaces, van Wijk recently extended his IBFV[14] to IBVFS[3] which first advects the mesh vertices in 3D space, and then projects the vertices to the screen to advect a screen space aligned input texture. Laremee *et al.*proposed another image space-based method called ISA,[4] which projects the mesh vertices as well as the vector field to the 2D screen before texture advections are performed. Weiskopf *et al.*[16] proposed a unified framework for 2D time-varying fields that can generate animated flow textures to highlight both instantaneous and time-dependent flow features. Recently, Li, Bordoloi, and Shen proposed a Chameleon algorithm[17] utilizing GPU-based dependent texture hardware for a more flexible control of texture appearance to visualize 3D steady and unsteady flows.

For 2D curvilinear meshes, Forssell proposed to perform the line integral convolution in the computational space.[1] Additional steps were taken to ensure a correct animation speed on the surface by adjusting the convolution kernel length based on the grid density in the physical space. To tackle the texture aliasing problem, Mao *et al.*[2] proposed to use multi-granularity noise texture based on a stochastic sampling technique called Poisson ellipse sampling. Although the problem of aliasing in grid of high density is alleviated, their method is not interactive and cannot adapt to arbitrary viewing conditions as the user zooms in and out of the field. Both methods are software based approaches and thus are neither interactive nor taking advantage of the capability now available in the programmable graphics hardware.

3. ALGORITHM

The primary goal of our research is to visualize 2D steady and unsteady flow fields defined on recti- or curvi-linear surface meshes that exist in 3D space. An example of such data is a computational plane from a curvilinear mesh as shown in Figure 7 presented in the result section 5. To provide the interactivity at run time, texture advection in our algorithm is computed directly at each fragment in the image space using graphics hardware. Particle paths and the input texture that will be advected are defined in the object space to generate accurate and perceptually correct appearances. For grid cells that cover multiple pixels, our goal is to generate flow patterns with enough details within the projection region of those cells in the output image without up-sampling the flow field or integrating additional particle paths.

The main idea behind our algorithm is that in order to generate textures of various resolutions to match the screen resolution as the user zooms in and out of the data, it is important to have an intermediate representation for the flow field whose resolution can be easily adjusted at run time to allow flexible and efficient texture advection. This intermediate representation should avoid the problems commonly encountered when up- or down-sampling a flow field: up-sampling the vector field would create a large overhead for computing additional particle traces, while down-sampling the vector field can generate incorrect flow paths. The latter is because a small error in each vector from the down-sampled field can accumulate to a large error as the numerical integration is computed. Another important criterion for devising our algorithm is that such an intermediate representation should be used directly by the texture advection algorithm and allows for an effective use of modern graphics hardware. In the following, we present our algorithm in detail.

3.1. Trace Slice

The core of our algorithm is a novel representation of the underlying flow field, called *Trace Slice*, which is used for multiresolution texture advection under different viewing conditions. Instead of generating multi-resolutional vector fields and using the approximated field to perform texture advection, the advection paths of flow textures can be more accurately obtained at arbitrary resolutions using the trace slices. The trace slices also allow us to exploit programmable GPUs to perform view dependent texture advection for each fragment at an interactive speed. In the following, we first present the concept of trace slices.

A trace slice $S_{T_s}^{T_d}$ is a 2D array with the same dimension as the input grid, which is essentially a parametric surface. Each element of the trace slice corresponds to a mesh vertex, which can be indexed by coordinates (i,j) defined on the parametric surface. The attribute stored in the trace slice, denoted as $S_{T_s}^{T_d}(i,j)$, is a 2-tuple (a,b), which means if a particle is released from (a,b) in the flow field at time step T_s, it will reach vertex (i,j) at time step T_d, where $T_d > T_s$. In essence, the information stored in a trace slice $S_{T_s}^{T_d}$ is primarily used to advect a texture N released at time step T_s to time step T_d. This is done by using the 2-tuple (a,b) stored in $S_{T_s}^{T_d}(i,j)$ as the texture coordinates to look up the input texture N defined at T_s.

To create the trace slices for a given 2D flow field, for every grid point (i,j) in the mesh, we perform the following steps for each time step $T_d = 0..T_{max}$, where T_{max} is the maximal time step in the time-varying field. Given a time T_d, we first perform a *backward* advection from the grid point (i,j), which will result in a pathline travelling in the space-time domain. Then, we sample the pathline at a sequence of time instants $t_i = T_d - i \times \Delta t, i = 1..K$ to get K positions of the pathline for a constant K when t_i stays greater than zero. Here we defer the discussion about the choice of K to section 3.3. Since the pathline is advected backwards, this sampled space-time position (a,b,t_i) implies that a particle released from (a,b) at time step t_i will travel forwards and arrive at (i,j) at time T_d. According to the definition of trace slices, the position (a,b) is stored into $S_{t_i}^{T_d}(i,j)$. If we repeat the same process for all grid points from the input mesh, a two-dimensional array $S_{T_i}^{T_d}(i,j), i \in [0,I_{max}], j \in [0,J_{max}]$ is formed, which is a trace slice. Because we sample the pathlines K times for different t_i, we have K trace slices which all have the same destination time step T_d. If the underlying field is a steady flow field, we use the virtual time typically used for computing streamlines in this process. Figure 1 illustrates the process of creating K trace slices.

3.2. Texture Advection

We now describe how the trace slices are used to perform flow texture advections at run time for a fixed resolution. Multiresolutional texture advection will be described in section 4. Since the underlying flow field is defined on a 2D structured recti- or curvi-linear mesh, we can render this mesh surface using one (for a regular Cartesian grid) or multiple

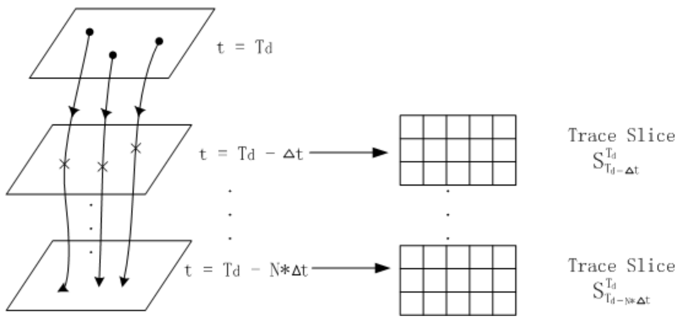

Figure 1. The creation of trace slices by backward advection

polygons (one per cell for a structured curvilinear grid) mapped with flow textures that we compute at run time. Our texture rendering algorithm takes as input an initial texture that is to be advected, and a set of trace slices loaded in as 2D textures. Without a loss of generality, in the following we assume the texture to be advected is a noise texture, although any texture can be used as an input for the advection.

One information needed is a mapping from the trace slice to the mesh surface. This is necessary because when the mesh geometry is rasterized, each fragment will look up the trace slices and use the 2-tuples stored in the corresponding locations to access the noise texture. In our algorithm, this mapping is established using the 2D mesh parameters (i,j), $i \in [0, I_{max}]$, $j \in [0, J_{max}]$ as the texture coordinates for the mesh vertices, where I_{max} and J_{max} are the dimensions of the 2D structured mesh.

Besides the mapping between the trace slices and the surface mesh, we also need a mapping from the input noise texture to the mesh. Conceptually this mapping can be seen as distributing the noise on the surface before advections take place. In our algorithm, when the 2D mesh is a regular Cartesian grid, the mapping is the same as how we map the trace slices to the mesh surface. For a 2D curvilinear mesh, however, care should be taken so that the noise is mapped to the 2D mesh in the physical domain as uniform as possible regardless of cell sizes and shapes.

With the mapping from the trace slices and the input noise to the mesh being established, the texture advection algorithm can now be explained as follows. Given an input texture N released at time T_s, knowing that the texture color at (a,b) from time step T_s will be advected to the point (i,j) at time step T_d if the trace slice $S^{T_d}_{T_s}(i,j) = (a,b)$, we can perform a two-stage texture look-up to advect the texture N to time step T_d. First, the mesh polygon is rasterized, where each fragment will interpolate the texture coordinates provided to the surrounding mesh vertices and then look up the trace slice texture using the interpolated texture coordinates. Then, the 2-tuple (a,b) retrieved from the trace slice texture $S^{T_d}_{T_s}$ for the fragment is used as the texture coordinates to look up the input noise texture N. We can express the advection of the noise texture from T_s to T_d as:

$$C(i,j,T_d) = N(a,b) = N(S^{T_d}_{T_s}(i,j)) \tag{1}$$

where $C(i,j,T_d)$ is the color for the fragment with the texture coordinates (i,j) at T_d, (a,b) is the trace slice 2-tuple stored at (i,j) in the trace slice, and $N(a,b)$ is the texel from the noise texture. Figure 2 illustrates our 2-stage texture lookup algorithm for texture advection.

3.3. Spatial Coherence

The advection algorithm presented above only calculates the influence of the input texture released at T_s to the frame at T_d. In fact, a fragment at time T_d will receive contributions from the noise texture released at multiple time steps. That is, the output color for each fragment at time T_d is a combination of input textures released earlier, which can be written as:

$$C(i,j,T_d) = \sum_{i=1..K} \frac{N(S^{T_d}_{T_d - i\times\Delta t}(i,j))}{K} \tag{2}$$

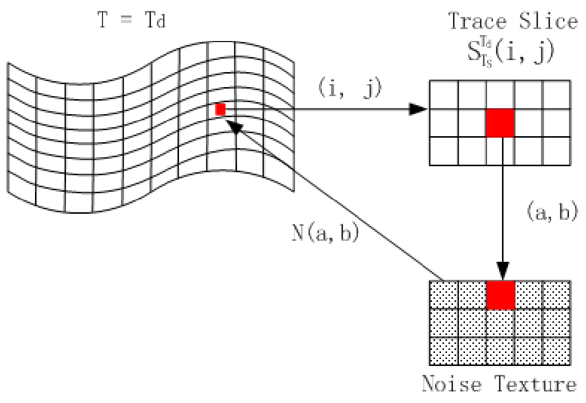

Figure 2. Texture advection using 2-stage texture lookups

where K is the number of contributions from textures released in the earlier time steps before T_d. An average of the color contribution from the K input noise texture is assigned to the fragment.

As demonstrated by the LIC algorithm,[7] a coherence of the pixel intensity along the flow lines provides effective cues to illustrate the underlying flow direction. The combination of textures described here allows us to establish such pixel intensity coherence along the pathlines that are relatively steady. This is because adjacent pixels along the pathline will have a large overlap in their backward advection traces.

Note that our method to create the texture advection result at each animation frame T_d is different from some of the existing methods (LEA,[15] IBFV,[14] IBFVS[3]) in the sense that there is no need to use the output of the previous frame as the input texture to compute the current frame. There are several reasons for this choice. First, the removal of inter-frame dependency allows the user to change the camera position or transform the mesh surface continuously since now the consecutive frames do not need to be rendered under the same view, a requirement for the previous methods. Second, when the underlying domain is not a simple 2D flat plane and hence occlusions may occur, it is not possible to transform the output from the frame buffer back to the object space and continue the advection for the next frame. Furthermore, as will be described in the next section, our algorithm computes the output image using different resolutions of trace slices $S_{T_s}^{T_d}$ and input texture N based on OpenGL mipmapping mechanism. The resolution of the trace slice and noise texture to use is determined for each fragment independently. Therefore, unless we compute the texture advection for the previous frame at all levels, which is expensive, it is difficult to satisfy the need of all fragments that may be rendered at different mipmap levels. Finally, the animation frames in our algorithm can be generated simultaneously, so it becomes possible to use different threads with multiple graphics cards to implement our algorithm when the underlying data set is large.

4. MULTI-RESOLUTIONAL TEXTURE ADVECTION

In essence, the motivation for generating multiresolution textures is to address the problems of aliasing and a lack of detail when visualizing the flow fields. With the trace slices and the texture advection algorithm presented above, our algorithm can generate multiresolution flow textures under various viewing conditions by adjusting the resolutions of the input noise and the trace slices on the fly. Specifically, to avoid the rendering artifact, it is important to ensure that the ratio between the size of a texel from the input noise as well as the trace slice to the size of a fragment within the object's projection area on the screen remains approximately one. This requirement can be enforced by using the classic *mipmapping* algorithm.

In our algorithm, we exploit OpenGL's mipmapping function to implement the idea of multi-resolutional texture advection. Starting from a base level of the input noise texture at a pre-defined resolution, we average every 2x2 texels recursively to create a sequence of mip-mapped input textures. The same operation will be applied to each of the trace slices, which is equivalent to creating a multi-resolutional version of the particle trace locations. We note that although conceptually creating a lower resolution of a trace slice is similar to down-sampling the flow field, it is in fact fundamentally different in the sense that our trace slices are computed from the original field. Integrating the particle using a vector field of a lower resolution is much more susceptible to accumulating errors, which makes the pathline drift away from the correct path.

As the user zooms out of the field, when the density of the projected mesh cells exceeds that of the screen pixels, OpenGL mipmapping will be triggered which will automatically choose an lower resolution of trace slice for each fragment to perform the texture advection algorithm presented above. An appropriate level of the input noise texture will also be chosen when the trace slice 2-tuple $S_{T_s}^{T_d}(i,j)$ is used to access the noise texture. In our algorithm, mipmapping mainly helps when multiple cells are projected to a single pixel to prevent from generating aliased textures. When a cell is projected to multiple pixels, on the other hand, since our texture advection is computed at per fragment level, flow patterns of fine granularity within the cell will still be generated. This is because each fragment within the cell will look up the trace slice based on the texture coordinates interpolated from the corners of the cell and perform the advection of noise texels along the interpolated pathline locations. As long as the input noise texture has enough resolution, the resulting flow texture will convey discernable flow patterns.

4.1. Adjustment of Advection Step Size

According to equation 2, each output fragment will take a sequence of samples from the input noise texture following the pathline traces. To avoid aliasing and ensure spatial coherence between adjacent fragments along the same pathline, it is important for each fragment to sample contiguous texels from the input noise. This will allow adjacent fragments along the pathline to average a similar set of noise input, hence creating spatial coherence. Previously, van Wijk and Jobard[14, 15] have made similar observations and suggested that the step size for the particle integration should satisfy the following rule:

$$|v| * \Delta t <= w \tag{3}$$

where v is the velocity at the current fragment location, Δt is the step size, and w is the texel width.

In our algorithm, Δt in equation 2 should be adjusted based on the resolution of the noise texture used for the fragment decided by OpenGL's mipmapping algorithm. However, since the level of detail for each fragment is determined by OpenGL independently at run time and not directly known to the application program, it is more difficult to determine Δt for each fragment from our program. Fortunately, since Δt is proportional to the texel size, and hence related to the noise texture resolution, we can compute a set of mipmapped Δt to accompany with the mipmapped noise texture. To do this, for the noise texture at the highest resolution, we first map the texel size to the space where pathines are computed. Then, we compute a Δt for each grid point based on its local velocity. This will produce a 2D array of Δt at the same resolution as the noise texture. Then, we create a sequence of down-sampled Δt textures in a similar manner as we create other mipmaps, except that now at each level, we will need to multiply the down-sampled Δt values by two since the corresponding texel size in the corresponding noise texture when mapped to the mesh surface becomes twice as large in each dimension every time we reduce the resolution by one level. Once the Δt mipmaps are created, at run time, we will use the same texture coordinates for the noise texture to look up the mipmapped Δt slices. Since the Δt texture has the same resolution and the same number of mipmapping levels as the noise texture, each fragment will use an appropriate Δt to match the noise texel size w.

According to equation 2, Δt values accessed from the mipmapped texture will be used to access the trace slices. It is possible that the value $t_i = T_d - i \times \Delta t$ in the equation will fall in between the trace slices that we have sampled. In this case, a linear interpolation from adjacent trace slice 2-tuples will be performed in our fragment program before looking up the noise texture.

5. RESULTS AND DISCUSSIONS

We have implemented our algorithm using OpenGL 1.5 and OpenGL Shading Language (GLSL) 2.0 running on a PC with an Intel Pentium 4 2.00 GHz processor, 768 MB memory, and an nVIDIA 6800 GT graphics card with 256 MB of video memory. The flow advection algorithm described above is primarily implemented in a fragment program. Each fragment is provided with the texture coordinates used to access the trace slices. The textures input to the fragment program include the noise texture, the Δt texture, and K trace slices where K is the convolution kernel size according to equation 2. The mipmaps for all the input textures are implicitly managed by OpenGL run-time system, so do not need special handling in the fragment program.

Three data sets were used to test our algorithm, as listed in Table 1. The vortex data set is a time-varying regular Cartesian grid data, and the rest are steady curvilinear grid data. We computed the trace slices by first starting a backward pathline at every time step from each grid point, and then sample the backward pathline locations. We let each pathline go

Data Set	Dimensions	Total Size
Post	32x76	28.5
Shuttle	52x62	37.7
Vortex	100x100	3747

Table 1. Datasets used in our experiments. Note that the size for the vortex data includes all 31 time steps. The sizes are in KBytes.

backwards as far as K time steps, where K is equal to the convolution size described in equation 2. When the underlying data set is a steady data set, the pseudo time for the particle integration is used. In all of our experiments, K was set to 10. The value of K would affect whether our convolution algorithm can be completed in a single pass or not. If K exceeds the maximal number of active textures that is allowed by GLSL, we need to split the texture advection into additional passes. The experiment results represented in this paper were all done in a single rendering pass.

In the process of computing the trace slices, if a particle goes out of bound before K time steps, we terminate it and the trace slice values for the remaining time steps are set to where the particle exits the domain. All trace slices were computed in a preprocessing stage. Table 2 column two lists the total preprocessing time for each data set. For a steady data set, the preprocessing time is within a few seconds. For the time-varying data vortex data, the preprocessing time is slightly larger. It is worth noting that the preprocessing only needs to be done once, and can be used for all output resolutions and different viewing conditions. The preprocessing also frees up the run-time algorithm from computing any particle trace, which significantly speeds up the run-time speed.

Data Set	Pre-Processing	Texture Creation and loading
Post	2.312	0.08
Shuttle	3.172	0.077
Vortex	8.875	0.24

Table 2. The time for trace slice preprocessing and texture creation and loading (in seconds)

When the user zooms in and out of the field, the levels of detail for both the trace slices and noise are adjusted automatically and the texture advection is computed on the fly. Using graphics hardware, this multi-resolution texture advection can be done very fast. For all the data sets we have, after the textures were loaded into the video memory, which only needs to be done once in the beginning of the program, the frame rate to advect and render the texture exceeded several hundred frames per second while the level of detail is being adjusted automatically. This is in fact not a surprise to us because the nVIDIA GeForce 6800 GT can render 5.6 billion texels, and 525 million vertices at each second. The amount of geometry and textures we had were considerably lower than the peak load that the graphics hardware can handle. Table 2 column three lists the time for creating and loading all the necessary textures to the video memory. Note that it is a process that only needs to be done in the beginning of the program so is part of the program set-up time.

We performed the following tests to verify the core idea of trace slices, that is, creating a down-sampled version of trace slice is more accurate than integrating particles using the down-sampled flow field. Using the vortex data set with a resolution of 100x100, we first created the trace slices $S_t^{10}(i,j), t \in [0,9]$ at a resolution of 100x100, and then down-sampled it to a resolution of 50x50 and a resolution of 25x25. We also down-sampled the flow field to 50x50 and 25x25 and computed backward pathlines using the down-sampled data. For the particle locations computed both from the down-sampled trace slices and the down-sampled fields, we compared them with the particle locations using the original field. Figure 3 shows the results for the 50x50 resolution. It can be seen that as the particles travelled farther, larger errors were accumulated when the down-sampled field was used. For the trace slices, the errors were bounded and no accumulation took place. As the data set was further down-sampled to 25x25, as shown in Figure 4, the error of the particle locations became larger when using the down-sampled field.

Figure 5(a) shows a snapshot of the Post data set with our multi-resolutional texture advection while Figure 5(b) shows a snapshot without such a control. The mesh of the Post data set has some interesting characteristics - the cells toward

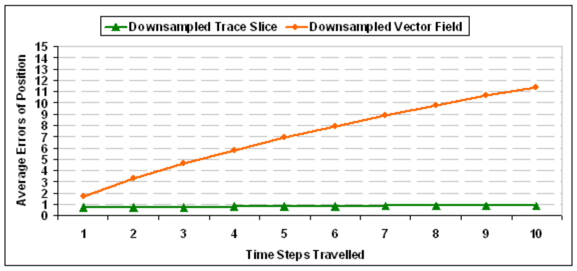

Figure 3. Comparison of particle position errors for travelling 1 to 10 time steps using the down-sampled trace slices and the down-sampled vortex data set reduced from 100x100 to 50x50. X axis indicates the time steps that the particles have travelled, and Y axis indicates particle position errors compared to the accurate traces using the Euclidean distance in the field.

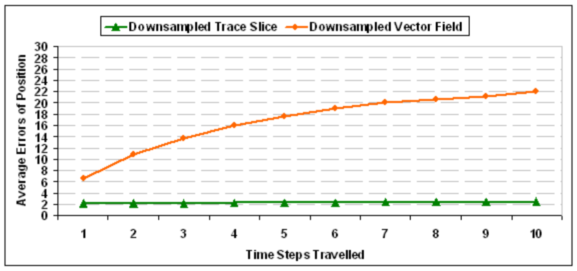

Figure 4. Comparison of particle position errors for travelling 1 to 10 time steps using the down-sampled trace slice and the down-sampled vortex data set reduced from 100x100 to 25x25. X axis indicates the time steps that the particles have travelled, and Y axis indicates particle position errors compared to the accurate traces using the Euclidean distance in the field.

the center of the mesh quickly become much smaller than the cells close to the outside boundary, with several orders of magnitude difference. It can be seen that our algorithm can generate a result with less aliasing.

Figure 6(a) and (b) show the comparison of the images for curvilinear mesh generated by our algorithm and traditional LIC. Both of them were generated at the original resolution of 52x62 for the portion that is shown. Our algorithm can generate clearer flow patterns without up-sampling the field, while LIC image in Figure 6(b) clearly does not include enough detail to show the flow directions.

Figure 7 shows the test results for the shuttle data set. Starting from a more close-up view and as we zoomed out, it can be seen from the image on the upper right that our multi-resolutional algorithm quickly switched to a lower resolution of traces and noises and thus still produce clear flow pattern. The image on the lower right did not use the multi-resolution algorithm. The same test was performed on the vortex data set. The image on the left of figure 8 is a close-up view. The image on the upper right is the result generated with our multi-resolution algorithm and the image on the lower right without. Similar to the shuttle data set, our algorithm was able to generate a clearer flow patterns since aliasing was avoided.

The overhead of our algorithm is the additional space for storing the trace slices. Table 3 shows the total size of the trace slices created for each data set. For the data sets that we tested in this paper, the overhead are all moderate. We believe there is still a large room to optimize the space consumption since there is a high degree of redundancy existing between trace slices of adjacent time steps. This will be our future work.

Data Set	Size of Trace Slices
Post	1.00
Shuttle	1.32
Vortex	21.8

Table 3. The size of trace slices (in MBytes) including all time steps. Note that Vortex data set is time-varying

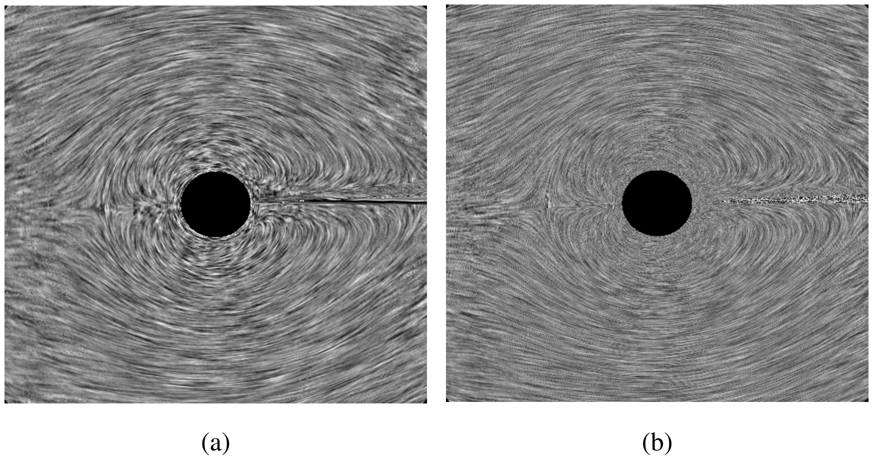

Figure 5. Rendering of the post data set (a) with and (b) without the multi-resolution level of detail control

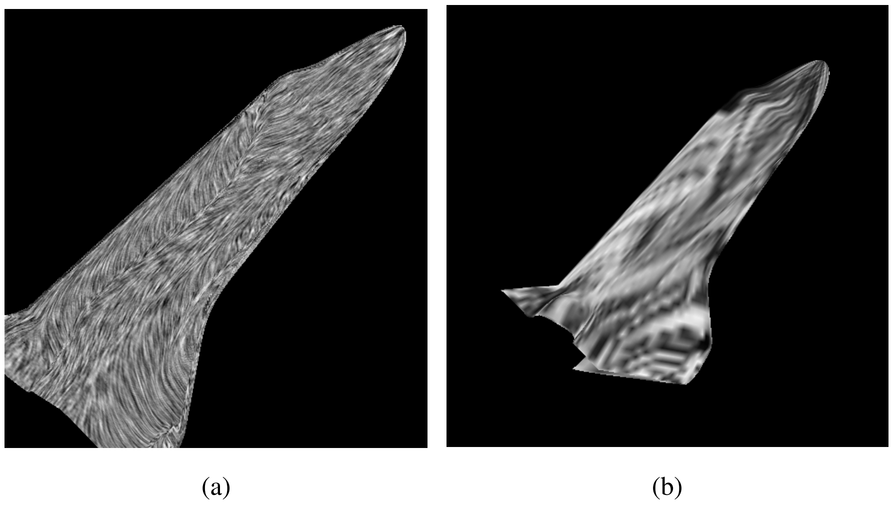

Figure 6. (a)With the correction of noise distribution, no stretched pattern can be seen (b) Rendering using LIC with the original resolution of 52x62.

6. CONCLUSIONS AND FUTURE WORK

We have presented a view-dependent technique for multi-resolution flow texture advection for 2D structured rectilinear and curvilinear grid data. Our algorithm can adjust the output texture resolution on the fly as the user zooms in and out of the field to prevent from aliasing as well as ensuring enough detail is presented in the image regardless of the mesh size and density. We propose a novel representation for the underlying flow field called trace slice which allows for flexible up- and down-sampling of the field without scarifying the accuracy or introducing additional cost for particle advections. Our algorithm is an image space method in that the flow texture is directly computed at each fragment. It is also an object space method because both the particle integration and the texture advection are computed in the object space. This trace slice representation can be directly used by GPUs to perform texture advection at a highly interactive speed.

The future work includes extending the proposed technique to three-dimensional data that allows the user to perform arbitrary zoom-in and -out. We will also optimize the space consumption for the trace slices since we believe there is a large degree of coherence between trace slices of adjacent time steps. Currently the amount of detail that we can provide within the mesh is not dependent on the resolution of the data, but the resolution of the input noise texture. We will investigate the use of procedural textures to perform the advection.

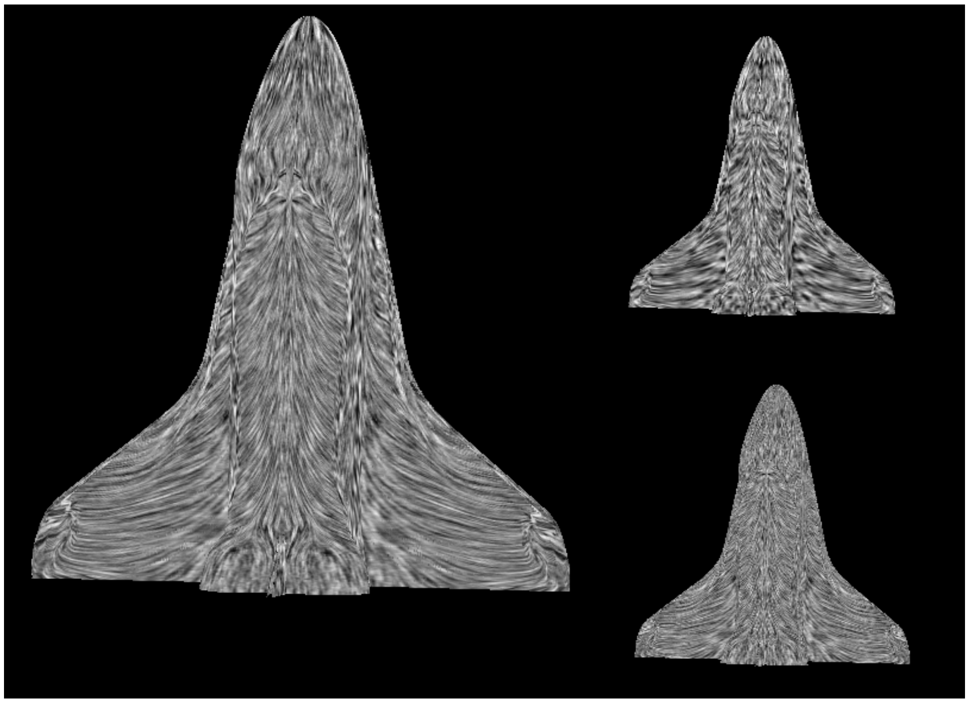

Figure 7. The image on the left was generated when we zoomed in. As we zoomed out from the image on the left, our algorithm was able to produce a clearer pattern by switching to a lower resolution of trace slices and noise texture (upper right), while the algorithm with no LOD control produced aliasing result (lower right).

REFERENCES

1. L. Forssell and S. Cohen, "Using line integral convolution for flow visualization: Curvilinear grids, variable-speed animation, and unsteady flows," *IEEE Transactions on Visualization and Computer Graphics* **1**(2), pp. 133–141, 1995.
2. X. Mao, L. Hong, A. Kaufman, N. Fujita, M. Kikukawa, and A. Imamiya, "Multi-granularity noise for curvilinear grid lic," in *Proceedings of Graphics Interface'98*, pp. 193–200, 1998.
3. J. van Wijk, "Image based flow visualization on curved surfaces," in *Proceedings of Visualization 2003*, pp. 123–131, IEEE Computer Society Press, 2003.
4. R. Laramee, B. Jobard, and H. Hauser, "Image space based visualization of unsteady flow on surfaces," in *Proceedings of Visualization 2003*, pp. 131–138, IEEE Computer Society Press, 2003.
5. G. Turk and D. Banks, "Image-guided streamline placement," in *Proceedings of SIGGRAPH '96*, pp. 453–460, ACM SIGGRAPH, 1996.
6. B. Jobard and W. Lefer, "Unsteady flow visualization by animating evenly-spaced streamlines," *Computer Graphics Forum (Proceedings of Eurographics 2000)* **19**(3), 2000.
7. B. Cabral and C. Leedom, "Imaging vector fields using line integral convolution," in *Proceedings of SIGGRAPH 93*, pp. 263–270, ACM SIGGRAPH, 1993.
8. J. van Wijk, "Spot noise: Texture synthesis for data visualization," *Computer Graphics* **25**(4), pp. 309–318, 1991.
9. D. Stalling and H.-C. Hege, "Fast and resolution independent line integral convolution," in *Proceedings of SIGGRAPH '95*, pp. 249–256, ACM SIGGRAPH, 1995.
10. H.-W. Shen and D. Kao, "A new line integral convolution algorithm for visualizing time-varying flow fields," *IEEE Transactions on Visualization and Computer Graphics* **4**(2), 1998.
11. M.-H. Kiu and D. C. Banks, "Multi-frequency noise for lic," in *Proceedings of the conference on Visualization '96*, pp. 121–126, IEEE Computer Society Press, 1996.

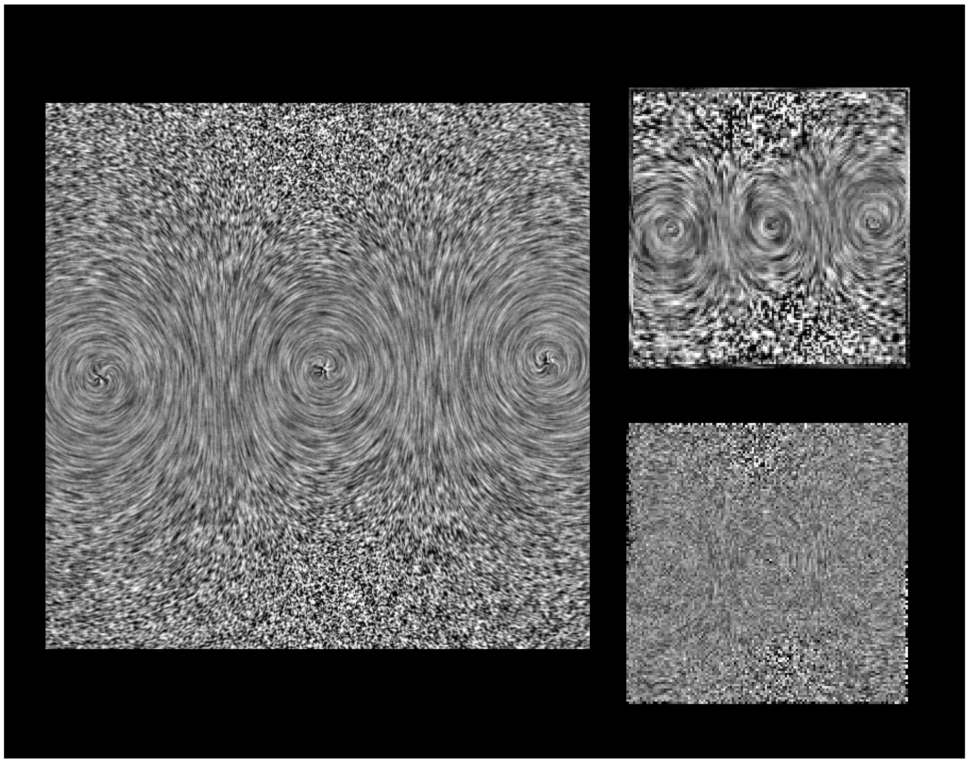
Figure 8. A similar test as Figure 7 using the time-varying vortex data set. It can be seen that our algorithm produced a better image (upper right) compared with no level of detail adjustment (lower right).

12. R. Wegenkittl and E. Gröller, "Fast oriented line integral convolution for vector field visualization via the internet," in *Proceedings of Visualization '97*, pp. 309–316, 1997.
13. H.-W. Shen, C. Johnson, and K.-L. Ma, "Visualizing vector fields using line integral convolution and dye advection," in *Proceedings of 1996 Symposium on Volume Visualization*, pp. 63–70, IEEE Computer Society Press, 1996.
14. J. van Wijk, "Image based flow visualization," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 745–754, ACM Press, 2002.
15. B. Jobard, G. Erlebacher, and Y. Hussaini, "Lagrangian-eulerian advection for unsteady flow visualization," in *Proceedings of Visualization '01*, pp. 53–60, IEEE Computer Society Press, 2001.
16. D. Weiskopf, G. Erlebacher, and T. Ertl, "A texture-based framework for spacetime-coherent visualization of time-dependent vector fields," in *Proceedings of Visualization 2003*, pp. 107–114, IEEE Computer Society Press, 2003.
17. G.-S. Li, H.-W. Shen, and U. Bordoloi, "Chameleon: An interactive texture-based rendering framework for visualizing three-dimensional vector fields," in *Proceedings of Visualization '03 (to appear)*, IEEE Computer Society Press, 2003.