A Surface LIC algorithm for parallel interactive exploration of large composite datasets

Burlen Loring

September 6, 2013

Abstract

1 Introduction

A composite dataset is simply a dataset organized into a tree of chunks of geometry and coresponding scalar, vector, and tensor fields. Because of their I/O scalability and the simplicity and flexibility in terms of load balancing when the number of visualization process differs from the number of processes used to compute a given simulation, composite datasets are employed ubiqutously in high perfromance parallel scientific data visualization. This work presents a surface LIC algorithm specifically tailored for interactive visualization and exploration of large composite datasets. By designing the parallel load balancing and compositing algorithm specifically for composite data gains in efficiency are attained by redcuing interprocess communication and redundant compuation.

The effectiveness of surface LIC technique is highly sensative to a number of external parameters such as the choice of surface geometry, the characteristics of the underlying vector field, the user's choie of camera postion, orientation, and other view related paramters, the user's choice of scene lighting, and screen resolution. The effectiveness of the technique is also affected by a number of internal paramters such as the characteristic of the noise used in convolution, and integration parameters such as step size and number of steps taken. By its nature the LIC process inherently reduces dyanmic range of its output. In the limit as the number of convolution steps approaches infinity the LIC output approaches a median(with respect to the input noise) uniform gray scale value. As this limit is approached patterns in the LIC are lost and ability to effecively visualize scalar field by applying color to LIC is diminished. As a user interactively explores a dataset varying any one of these parameters can drasticlly alter the result, thus a great deal of flexibility is needed to provide acceprtable result generally for all values of this paramter space. AUrtomatic and user tunable features that result in excellent results across the paramter space are presented.

The work presented here has been incorporated into the popular scientific data visualizaiton library VTK. It has also been specifically designed to work with the popular parallel compositing library IceT, so that it may also be used within popular parallel scientific data visualization codes like ParaView and VisIt which both use IceT for image compositing. The work is distributed with ParaView as well.

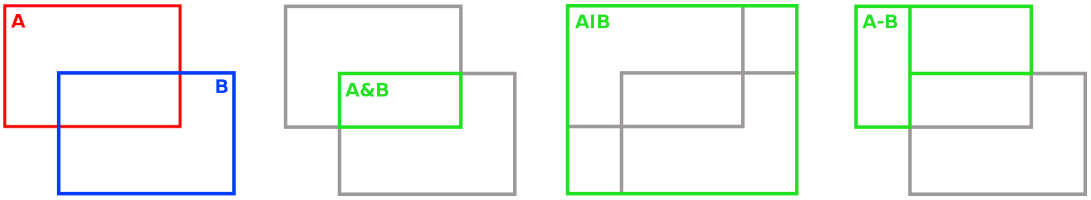


Figure 1: Operations on pixel extents. The results of three common operations on the two extents A, and B, are shown in green. Left, intersection. Center, union. Right, subtraction.

2 Preliminaries

2.1 Operations on pixel extents

When thinking about algorithms designed for rendering parallel composite datasets it's convinient to introduce the concept of a "pixel extent". A pixel extent is defined simply as a set of 4 integer values decribing an axis aligned bounding rectangle in screen space. This light weight meta data is useful for describing and manipulating regions ins screenspace. For example we use this abstraction to describe the view and subsets of the view containing valid data for each leaf dataset on each MPI process. Pixel extents are light enough to be quickly communicated to all processes with little overhead making it possible for all processes to know how data is distributed. Additionally, pixel extents are mapped easily to the MPI subarray datatype and thus simplyfy the description of "zero-copy" data transfers of non-contiguous regions of memory between processes or locally from one buffer into another. Here the source and destination buffers need not be the same size, or dimensions, but the source and destination extent do. Thus they facilitate transfers of data needed for load balancing and compositing.

When combined with mathematical operators pixel extents are a powerful abstraction for determining both the cost of and the steps required to composite the vector field. The following set of opertions are useful, sizeof, union($\cup$), intersection($\cap$), shift, merge, and subtract(-). For example in figure 1 given two overlaping extents A and B, if A and B are on different processes then during compositing the data described by the exent $A \cap B$ must be excahnged between these processes and communication cost of this exchange will be proportional to sizeof($A \cap B$).

2.2 Pipeline

Here an overview of the pipeline used to compute

Conceptually our surface LIC algorithm projects vectors defined on an arbitrary surface onto the surface and then from world space into screen space where an image LIC is computed[**?**] [**?**]. When running in parallel the transition from from world space to screen space necessitates a compositing step that gathers pixels to the process that needs them. Because we are implementing our algorithm inside of a larger eco-system, depending on the load balancing algorithm selected a scatter stage that puts pixels back on their original processes may be needed. This step is primarilly dictated by IceT which determines which pixels are touched by a given process and composites only those pixels. Our algorithm is MPI parallel and through GPGPU programming techniques can be accelerated by GPU's where available and threading where GPU's are not available[**?**].

A schematic of the algorithm is presented in figure 2.This is shown on the left half of the diagram. In this figure, optional processing stages are shaded gray, we show chached textures as red parallelograms, and green double arrows indicate inter-process communication. The major processing stages of the surface LIC are 1) lighting computations and vector projection, 2) a gaurd pixel generation, load balancing, and compositing stage, 3) a stage computing the LIC in image space, 4) A scatter stage to move the data

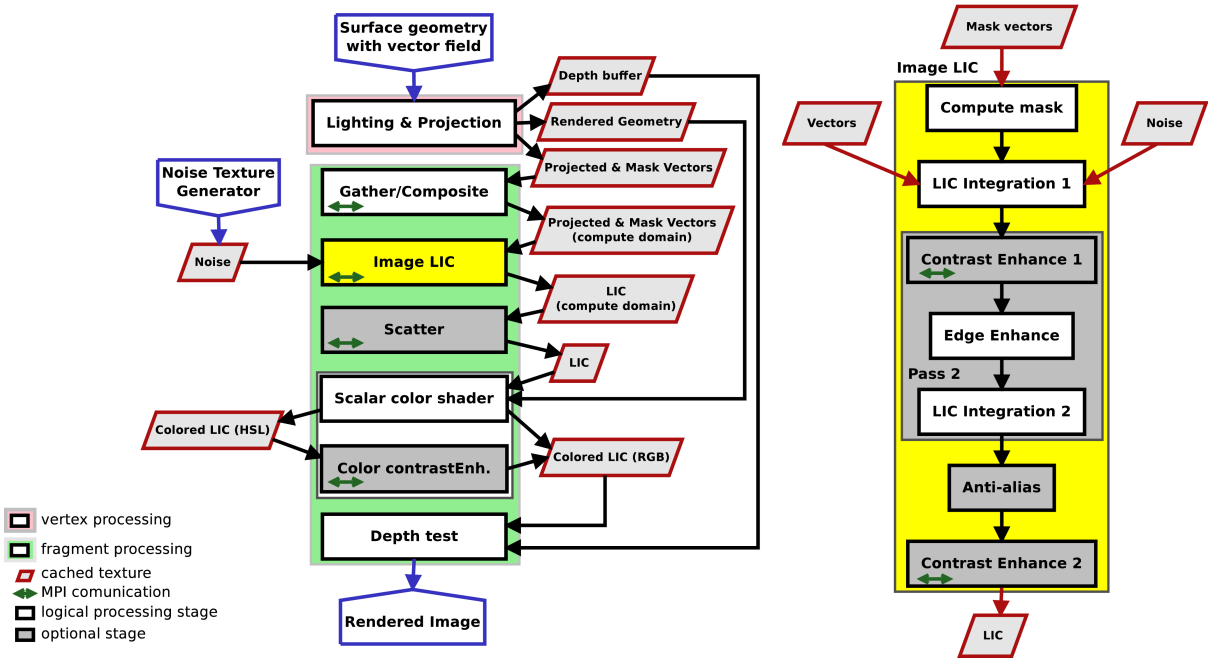


Figure 2: Logical processing stages

back to where it was prior to compositing; 5) a shading stage that combines image LIC and results of scalar coloring and lighting computations; 6) a color contrast enhancement stage; 7) a stage performing depth test and copying the final result into the default FBO. See appendix **??** for example of output of each stage.

We have split the algorithm into logical processing stages and show them organized in two groups, the vertex processing group(red) and the fragment processing group(green). During vertex processing we loop over each process's composite data leafs, rendering each to a single texture using a shader that projects vectors, computes lighting, and applies scalar coloring to lit surface geometry. During this stage depth test is enabled which results in a localy correct vector field in image space. To produce the globally correct vector field a compositing stage is employed. Prior to fragment processing we compute the screen space projection, or pixel extent, of each composite dataset leaf's world space axis aligned bounding box. Each leaf's pixel extent is further tightened using the depth buffer and vector field. We then utilize the collection of tight pixel extents to initiate the fragment processing stages by rendering screen aligned quads over just the fragments described be each pixel extent. Limitting fragment processing to tight pixel extents can significiantly reduce the number of fragments that each stage processes for example see section **??**.

On the right half of the figure a break-out diagram detailing the processing stages used in our image LIC algorithm are shown. Our image LIC algorithm is implemented in two passes. In the first pass, fragments are masked based on user provided criteria[**?**] then a basic LIC is computed[**?**]. In the second pass, the result of the first pass is run through contrast and edge enhancment stages before being used as "noise" texture in the second LIC pass. The second pass substantially improves the visual quality of the streaks. Finally optoinal anti-aliasing and contrast enharncement stages are applied.

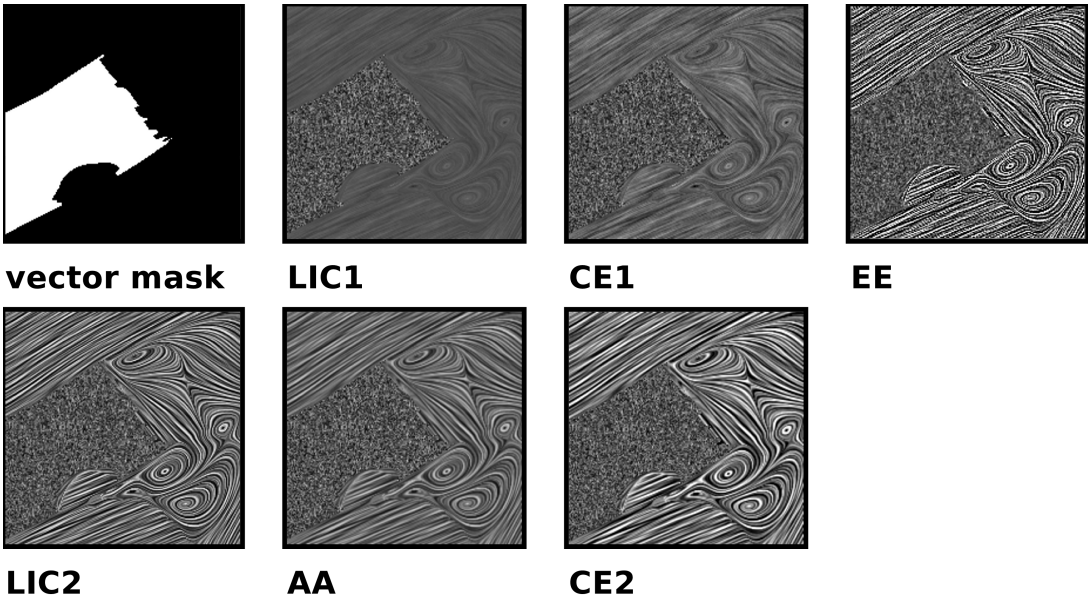


Figure 3: Line integral 2D stages

3 Features for interactive data exploration

What characteristics in the output LIC make for an effective visualization across a wide variety of input data and rendering conditons? Some of the important characteristics in an effective surface LIC visualizaion are

1. The LIC patterns accurately represent the characteristics of the underlying vector field. If desired the relative strength and features in strongly vearying fields should be maintained.
2. When scalar coloring is desired, LIC pattern must not be lost during the application of colors and similarly scalar colors once applied should not be dull or have greatly diminished intensity.
3. Over all a high dynamic range in the resulting image is desirable. This make it easy to identify feature in LIC and scalar colors. The lower the dyanmic range the less decernable detail there is in the result.
4. Lighting characteristics must be maintained, for example shadows, specular reflections, etc.
5. The algorithm should perform well on very large surface geometry

When thinking about how best to acheive these goals it's important to consider how scalar colored, lit geometry and and image LIC are combined to produce the final result. In our algorithm scalar coloring and lighting caclulations are rendered and stored in a texture. The image LIC is computed using projected vectors and also stored in a texture. There are two common approaches for combining these two textures into the final resut, a multiplciative or mapping approach[**?**] and an addative or blending approach[**?**].

The mapping approach is described by the following equation:

$$c_{ij} = (L_{ij} + f) * S_{ij} \tag{1}$$

where the indices i, j identify a specific fragment, c is final RGB color, L is LIC gray scale intensity, S is the scalar RGB color, and f is a biasing parameter, typically 0, that may be used for fine tuning. When $f = 0$, the typical case, colors are transfered directly to the final image where the LIC is 1, and a linearly scaled transfer of scalar colors where LIC gray scale color is less than one down to 0, where the final color is black. The bias parameter f may be set to small positiver or negative values between -1 and 1 to increase or decrease LIC values uniformly resulting in brighter or darker images. When $f \neq 0$ final fragment colors, c, are clamped such that $0 \leq c \leq 1$.

With the mapping approach the distribution of intensity values in the LIC directly affect the accuracy and intensity with which scalar colors and lighting effects are transfered to the final result and the average brightness and contrast of the result. Note in the final result that individual RGB channel values will be less then or equal to the maximum grayscale value in the image LIC. Also, the greater the number of pixels close to 1 in the image LIC, the more acurately and intensly scalar coloring and lighting are transfered into the final image. However, this must be balanced with a sufficient number of highly contrasting pixels where the value is closer to 0 in order to accurately represent the LIC pattern. Put sucintly it's critical that the image LIC has high contrast and dynamic range with a good mix of light and dark values if it is to be effectively mapped onto scalar colors. However, the convolution process inheriently reduces both contrast and dynamic range in the image LIC. To correct this we've introduced contrast enhancment stages in three places in the pipeline.

The blending approach for combining scalar colors, lighting effects, and the image LIC is described by the following equation:

$$c_{ij} = L_{ij} * I + S_{ij} * (1 - I) \tag{2}$$

where the indices i, j identify a specific fragment, c is final RGB color, L is LIC gray scale value, S is the scalar RGB color, and I is a constant ranging from 0 to 1, with a default of 0.8. Decreasing I to obtain brighter colors diminishes the intensity of the LIC, and vise versa. When colors are bright the LIC is difficult to see. Currently, the best results are obtained by sacrificing slightly on both fronts. The blending approach is especially useful with curved surfaces and pronounced lighting effects. Like the mapping approach it bennefits from an image LIC with high contrast and dyanmic range.

3.1 Noise generator

Important factors in determining the characteristcs of streaking patterns in the LIC and in overall contrast and dynamic range of the final image are the characteristics of the noise to be convolved with the vector field. By varying the properties of the input noise such as distribution, or noise type, minimum, maximum values, number of realizable values, size of each elemental noise value. spacing between noise values and background color of , the size of the noise texture, the contrast and dynamic range of the noise texture, all play a role in the look of the final image. Using the same noise texture produces markedly different results on different datasets and even on the same dataset at different screen resolutions. It's unlikely that a single noise texture will work well in all cases. Thus a noise generator is a key component in the algorithm for interactive data exploration.

Our noise texture generator defines the following 9 run time modifiable degrees of freedom.

Noise type This parameter controls the underlying statistical distribution of values in the generated noise texture, or type of noise generated. The user may choose from Gaussian noise, uniformly distributed noise, or Perlin noise. By default Gaussian noise is used.

Texture size This paramter controls the size of the square noise texture in each direction. Support for non-power of 2 textures is assumed. However in the case of Perlin noise the texture size is adjusted to the nearest power of 2.

Grain size Select the number of pixels in each direction that each generated noise element fills in the resulting texture. For Perlin noise this sets the size of the largest scale, and must be a power of 2.

Min value This parameter sets the lowest gray scale value realizable in the generated noise texture. This paramter can range between 0 and 1 and the default value is 0.

Max value This parameter sets the highest gray scale value realizable in the generated noise texture. This paramter can range between 0 and 1 and the default value is 0.8.

Number of levels Set the number of realizable gray scale values. This parameter can range from 2 to 1024 and the default value is 1024.

Impulse probability This parameter controls how likely a given element is to be assigned a value. When set to 1 all elements are filled. When set to some number lower than one a fraction of the texture's elements are filled with generated noise. Elements that are not filled take on a background color value. The default impulse probablity is 1.

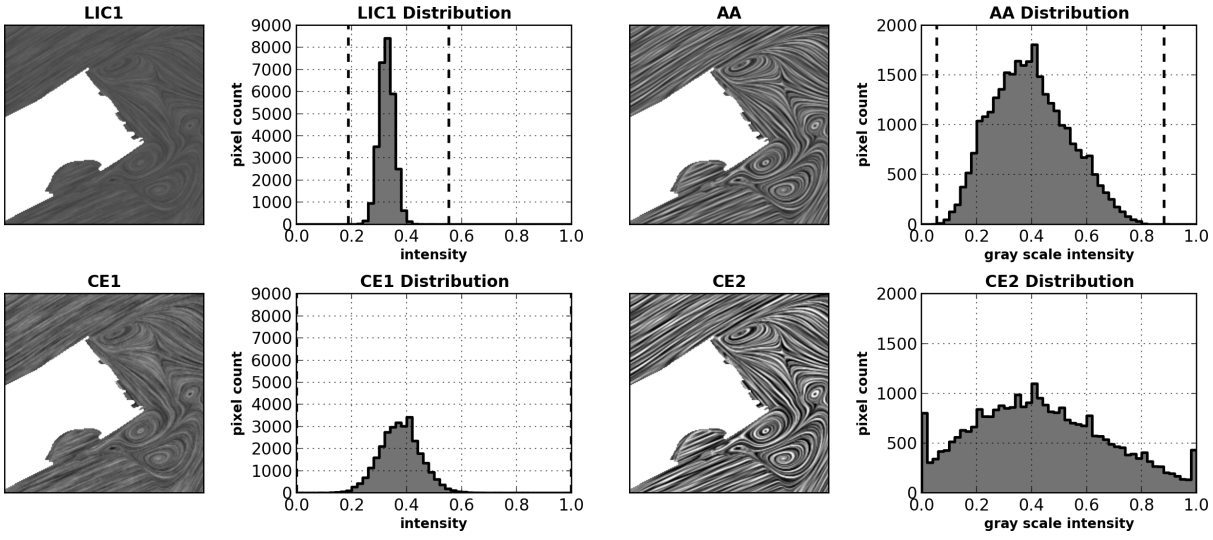
Background color The gary scale value to use for un-touched pixels when the impulse probability paramter is set less than 1. The default background value is 0.

RNG seed Modify the seed value of the random number generators.

Changing the noise texture gives one greater control over the look of the final image. TODO insert image.

3.2 Scalar color shading

3.3 Contrast enahancement



(a) (b)

The convolution process tends to decrease both contrast and dynamic range, often producing dark and dull results. Figure **??**.a illustrates this issue. The narrowing of dynamic range and diminishing of contrast are consequences of the convolution operation which convolves a noise texture with a vector

field. These reductions are made worse by the use of Gaussian noise because by definition centrally concentrates its values about a mid gray tone.

In order to counteract this, optional contrast enhancement stages have been added. The new stages increase the dynamic range and contrast without modifying the patterns that emerge from the LIC process or modifying the scalar colors. VTK's surface LIC implementation is provided by two classes, the vtkSurfaceLICPainter and vtkLineIntegralConvolution2D. The vtkSurfaceLICPainter, projects vectors, sets up noise texture, and passes control to the vtkLineIntegralConvolution2D to generate a LIC, after which it shades scalar colors combined with the generated LIC onto the surface geometry. Contrast enhancement stages have been added to both classes.

The contrast enhancement stages are implemented by histogram stretching of the gray scale colors in the LIC'ed image as follows:

$$c_{ij} = \frac{c_{ij} - m}{M - m} \tag{3}$$

where, the indices i, j identify a specific fragment, c is the fragment's gray scale color, m is the gray scale color value to map to 0, M is the gray scale color value to map to 1. When the contrast enhancement stage is applied on the input of the high-pass filter stage, m and M are always set to the minimum and maximum gray scale color of all fragments. In the final contrast enhancement stage m and M take on minimum and maximum gray scale colors by default but may be individually adjusted by the following set of equations:

$$m = min(C) + F_m * (max(C) - min(C)) \tag{4}$$

$$M = max(C) - F_M * (max(C) - min(C)) \tag{5}$$

where, $C = \{c_{00}, c_{01}, ..., c_{nm}\}$, are all of the gray scale fragments in the LIC image and F_m and F_M are adjustment factors that take on values between 0 and 1. When F_m and F_M are 0 minimum and maximum are gray scale values are used. This is the default. Adjusting F_m and F_M above zero controls the saturation of normalization. This is useful, for example, if the brightness of pixels near the border dominate because these are convolved less because we can't integrate outside of the dataset.

Occasionally, often depending on the contrast and dynamic range and graininess of the noise texture, jagged or pixelated patterns may emerge in the LIC. These can be reduced by enabling the optional anti-aliasing pass.

The color contrast enhancement stage is implemented with histogram stretching on the fragments lightness in the HSL color space.

$$L_{ij} = \frac{L_{ij} - m}{M - m} \tag{6}$$

where, the indices i, j identify a specific fragment, L is the fragment's lightness in HSL space, m is the lightness to map to 0, M is the lightness to map to 1. m and M take on minimum and maximum lightness over all fragments by default but may be individually adjusted by the following set of equations:

$$m = min(L) + F_m * (max(L) - min(L)) \tag{7}$$

$$M = max(L) - F_M * (max(L) - min(L)) \tag{8}$$

where, L are fragment lightness values and F_m and F_M are the adjustment factors that take on values between 0 and 1. When F_m and F_M are 0 minimum and maximum are lightness values are used. This is the default. Adjusting F_m and F_M above zero provides fine-tuning control over the saturation.

3.4 Integrator Normalization

Normalizing vectors during integration is a technique often used to simplify the LIC'ing algorithm and its use. During integration vector field values are normalized with the result that the convolution occurs over

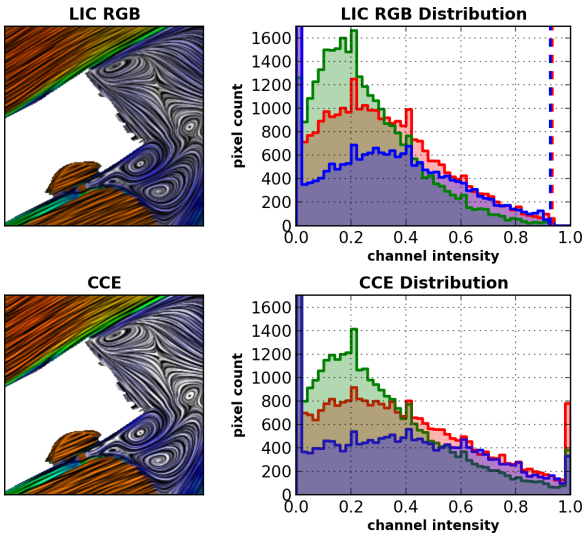


Figure 4: CE Curves

the same integrated arclength for all pixels in the result. This gives the result a smooth and uniform look independent of the input vector field. This also makes it possible to provide reasonable default values for step size and number of steps to the integrator independent of the input vector field. The resulting visualization acurately shows the true tangent field however variations in the relative strength of the vector field is lost and can alter the relationships of flow features making weak features prominent and strong features less so.

When normalized(the default) the input vector field is normalized during integration, and each integration occurs over the same arc-length. When not set each integration occurs over an arc length proportional to the field magnitude as is customary in traditional numerical methods. See, "Imaging Vector Fields Using Line Integral Convolution" for an example where normalization is used. See, "Image Space Based Visualization of Unsteady Flow on Surfaces" for an example of where no normalization is used. Both approaches are valid and useful, the new feature allows for the selection of one or the other.

In the context of developing a general puprose tool for interactive data exploration it's important to provide both options and let the user select the option that best fits her needs. For example figure 5 shows the result of the algorithm applied to a simulation of magnetic reconnection in a hot plasma with and without integrator normalization. In this case normalization can give a false sense of the importance of a number of flow features.

3.5 Fragment Masking

Currently the criteria for masking fragments is that all vector components must be identically zero. This doesn't work for many numerical simulations where stagnant flow is not identically zero due to numerical rounding. For example, stagnate flow might be where $|V| < 1e^{-6}$. Also, the lack of control over the color characteristics of the masked fragments result in masked fragments looking drastically different than the LIC'ed fragments. This is illustrated in figure **??**.d where all fragments on the outer walls have been masked , the masked fragments are much brighter than the the LIC'ed fragments.

New fragment masking implementation makes use of a user specified threshold value below which fragments are masked. The masking test may be applied either to the original vectors or the surface projected vectors. By applying the test to the original vectors the masked fragments will match scalar

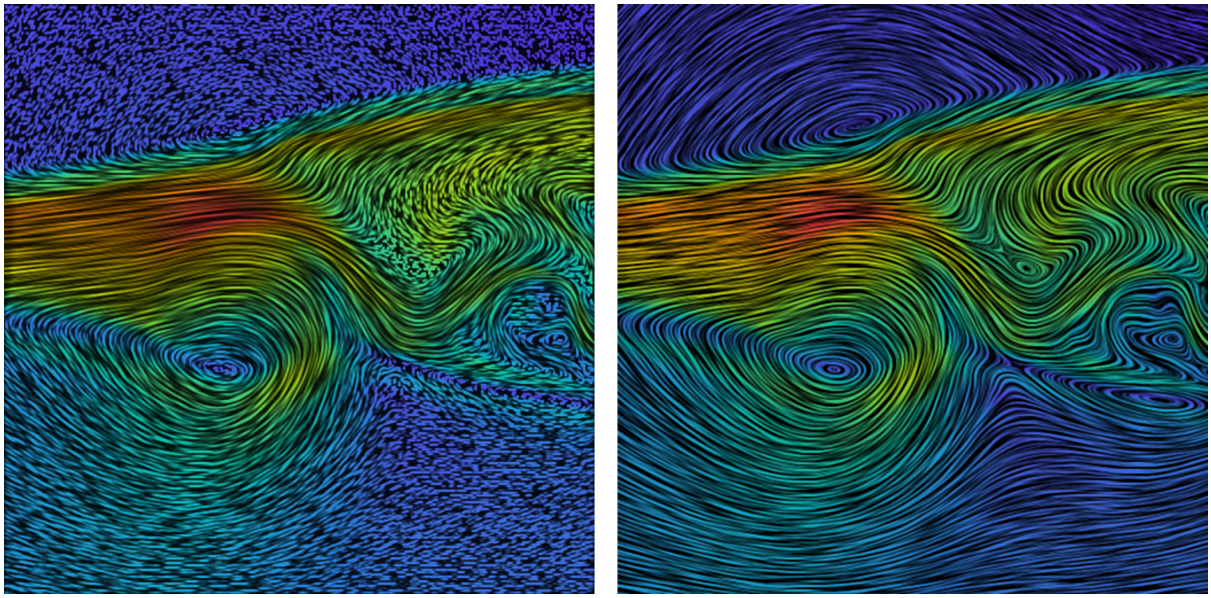


Figure 5: vector normal

colors when coloring by $|V|$.

Fragments where $|V| < t$ are masked. The fragment masking implementation provides control over the intensity and color of the masked fragments via the following equation:

$$c_{ij} = M * I + S_{ij} * (1 - I) \tag{9}$$

where the indices i, j identify a specific fragment, c is final RGB color, M is the RGB mask color, S is the scalar RGB color, and I is the mask color intensity. This allows one control over the masking process so that masked fragments may be: highlighted (by setting a unique mask color and mask intensity ¿ 0), made invisible with and without passing the un-convolved noise texture (by setting mask intensity 0), or made to show the scalar color at a similar level of intensity as the LIC (mask intensity ¿ 0).

3.6 Optimizations for interactivity

4 Parallelization

4.1 Gaurd pixel generation

To ensure correct parallel results prior to compositing guard pixel halos are added to compute pixel extents. The number of gaurd pixels required is determinied by the longest arc length of the integration on each pixel extent. When integrator noramlization is used the same across all pixel extents and is given by:

$$g = \max(n_s \delta n_p \alpha, 2) \tag{10}$$

where, g is the number of gaurd pixels required, n_s is the number of steps to integrate, δ is the step size, n_p is the number of image LIC passes (either 1 or 2), and α is an addative factor that compensates for beleding at the view edges where it's impossible to compute all n_s steps because guard pixel data is unavailable. At least 2 guard pixels are employed because linear texture fetches use a texels 4 nearest neighbors.

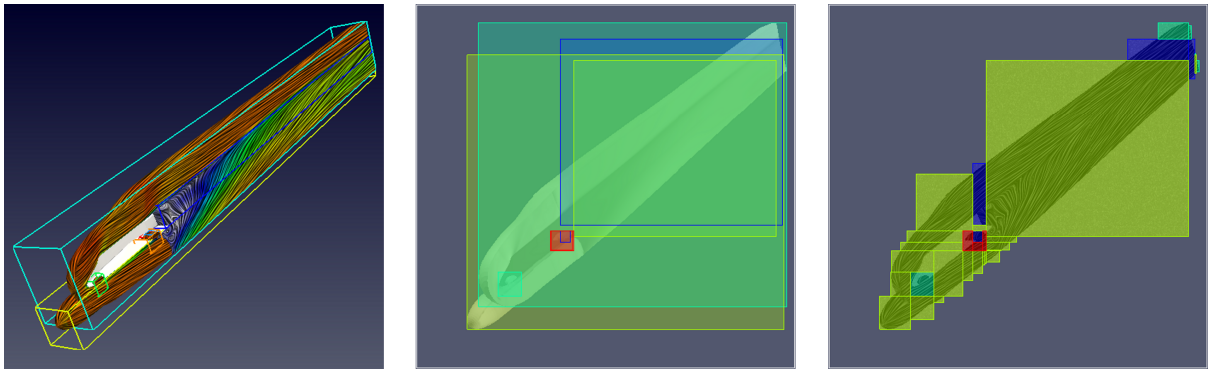


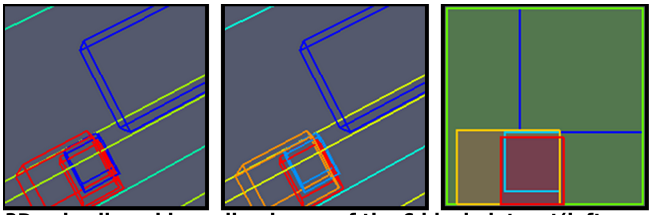
Figure 6: Left: LIC colored by momentum on a surface extracted from composite dataset simulating a launch vehicle[**?**] with world space axes aligned bounding boxes colored by dataset id. Center: Screen space projection of the composite dataset's axis aligned bounding boxes colored by MPI process id in a 4 way parallel run. For reference composited vector field colored by magnitdue is shown. The comm cost for this decomposition is 1.8. Right: Pixel extents from the in-place disjoint domain decomposition used to compute the LIC colred by MPI process id with the LIC blended for reference. The comm cost for this decomposition is 0.32.

Note that in **??** because vector field is normalized $n_s\delta$ is the arc length of the integration. Without integrator normalization the contribution of the vector field magnitude to the integrated arc length must be accounted for. An upper bound on field strength is required. One complication is that for a given pixel extent the maximum value on that extent may change as a result of compositing the data from the overlaping extents from other processes. This is solved for each extent by finding the maximum on itself and all overlapping extents. Note, that although we don't have the vector field defined on the set of comute extents, it is this set of extents that must be used. This is handled by searching for the maximum value inside the interesction of each overlaping rendering and compute extent.

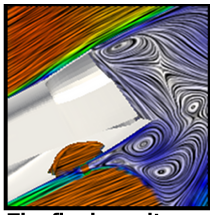
$$g_i = \max(n_s \delta n_p \alpha \max(\mathbf{V} \cap V_i), 2) \tag{11}$$

where V is a set, with members consisting of vector field data defined on each extent of the LIC computational domain, and V_i is the extent to compute the number of gaurd pixels for.

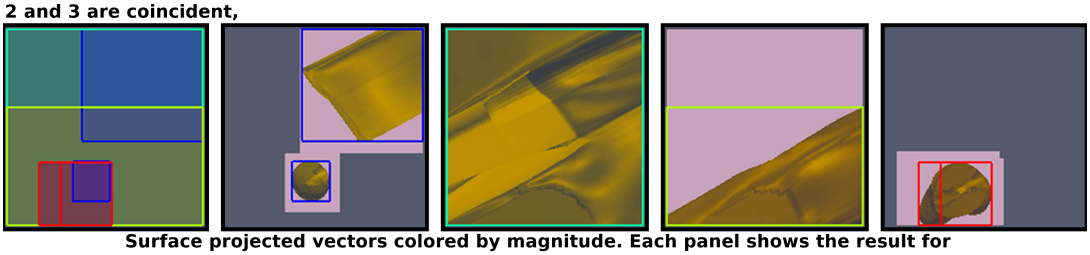
4.2 Compositing Algotithms



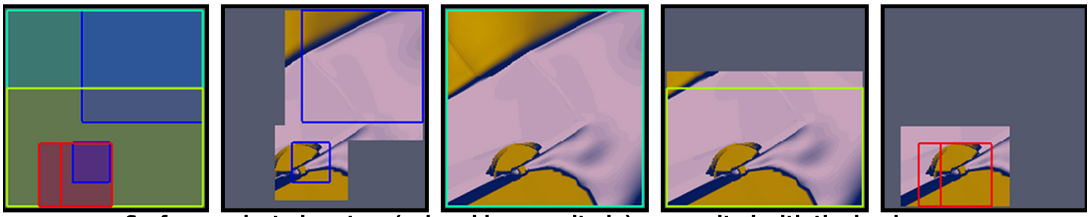
3D axis aligned bounding boxes of the 6 block dataset(left colored by MPI rank and center colored by block id) rendered on 4 MPI ranks, and coresponding 2D screen space pixel extents (right colored by block id). Note that the 2D projections of block 2 and 3 are coincident,



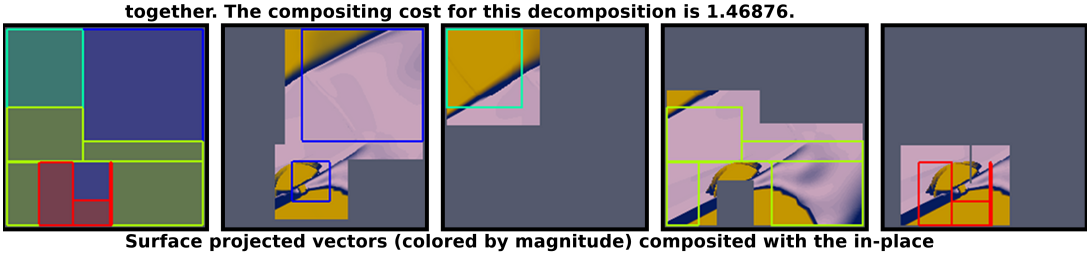
The final result.



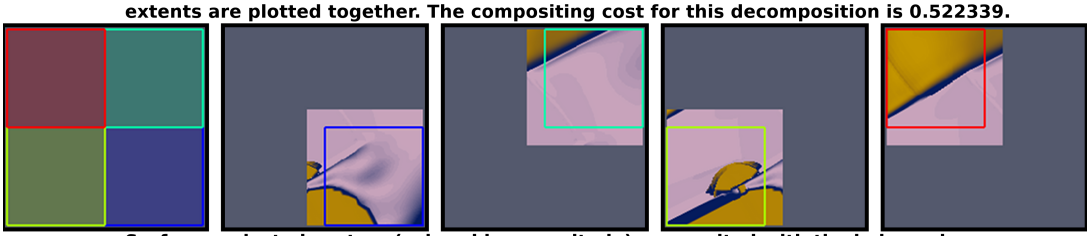
Surface projected vectors colored by magnitude. Each panel shows the result for one of 4 MPI ranks. Tightened pixel extents colored by MPI rank are superposed to show the savings attained by using tightened bounds. On the far left screen space extents for all ranks are plotted.



Surface projected vectors (colored by magnitude) composited with the in-place method. The screen space extents are superposed to show the domain where LIC is computed. The areas outside of the extents are gaurd pixels necessary for correct parallel computation. On the far left screen space extents for all ranks are plotted together. The compositing cost for this decomposition is 1.46876.



Surface projected vectors (colored by magnitude) composited with the in-place disjoint method. The screen space extents are superposed showing the domain where the LIC is computed. The pixels outside of the extents are gaurd pixels necesssary for correct parallel computaiton. On the far left the screen space extents are plotted together. The compositing cost for this decomposition is 0.522339.

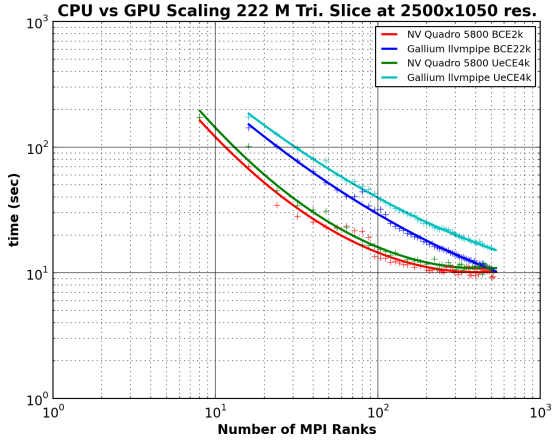


Surface projected vectors (colored by magnitude) composited with the balanced method. The screen space extents are superposed showing the domain where the LIC is computed. The pixels outside of the extents are gaurd pixels necesssary for correct parallel computaiton. On the far left the screen space extents are plotted together. The compositing cost for this decomposition is 0.522339.

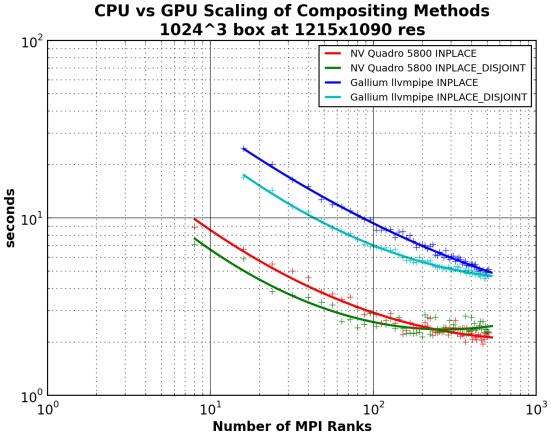
Figure 7: Compositing algorithms

4.3 Compositor Scaling

Over the past few days I've been analyzing the new parallel surface lic algorithms scalability and performance and I thought you might find the early results helpful as you look at the code. These runs were made with pvpython+pvserver on NERSC's Cray XC30 Edison using Mesa 9.2.0's OS Mesa llvmpipe driver with 4 rendering threads per rank and 8 ranks per node. There's also some documentation of the compositors in this talk (see the last few slides) http://www.hpcvis.com/vis/talks/rdav-ah-2013/rdav-ah-2013.pdf



(a) Test of CE scaling on two vector fields



(b) Compositor scaling on MHD turbulence dataset

Figure **??** shows scaling from 16 to 512 ranks on a massive 16384x8192 VPIC dataset. The figure shows runs on two vector fields using the settings for a high quality image with and without the new contrast enhancement feature. The figure shows that although the new CE feature makes use of MPI_Allreduce this does not negatively impact scaling. The differences in performance are attributable to the difference in number of integration steps. Typically I've found that to get the look I want I end up using more integration steps when CE is enabled. A sample image from this run is here: http://www.hpcvis.com/vis/vtk-surface-lic-parallelization/kh-new-jaguar-lic-b-woce.png

Figure **??** show scaling from 16 to 512 ranks on a large 1204^3 MHD turbulence dataset and a comparison of two of the compositing algorithms. The INPLACE option is what you and I had initially discussed on Skype, while the INPLACE_DISJOINT option ensures that each fragment is computed only once by uniquely assigning ownership of regions of the screen to ranks while leaving the data in-place ie. a rank doesn't take ownership of a fragments it doesn't have data for. For the INPLACE_DISJOINT approach the compositing communication cost is the same or in some cases less than the INPLACE approach although it's split into two phases(gather and scatter). The figure shows that the INPLACE_DISJOINT method offers a nice increase in performance. The speed up is larger in the smaller runs because in those runs the screen regions by rank are larger, and this presents a greater opportunity to reduce the number of fragments processed per rank. The following teo plots show this better. The image produced by these runs is here: http://www.hpcvis.com/vis/vtk-surface-lic-parallelization/pr1-lic.png

These two figures are Gant plots of each rank's activity during the 2 node 16 rank compositing INPLACE and INPLACE_DISJOINT scaling runs. We're looking at the 2-node case because it shows the greatest speed up for the INPLACE_DISJOINT compositor and it will be easier to see the performance differences between the two. For both runs most of the time is spent in RenderGeometry (vector projection and light vertex shaders) and Integrate1/Integrate2 phases. CE phases tend to look long because of the MPI_Allreduce ranks get synchronized by the slowest rank so most of the time spent there is waiting. The RenderGeometry and GatherVectors(compositing and guard pixel exchange) times are

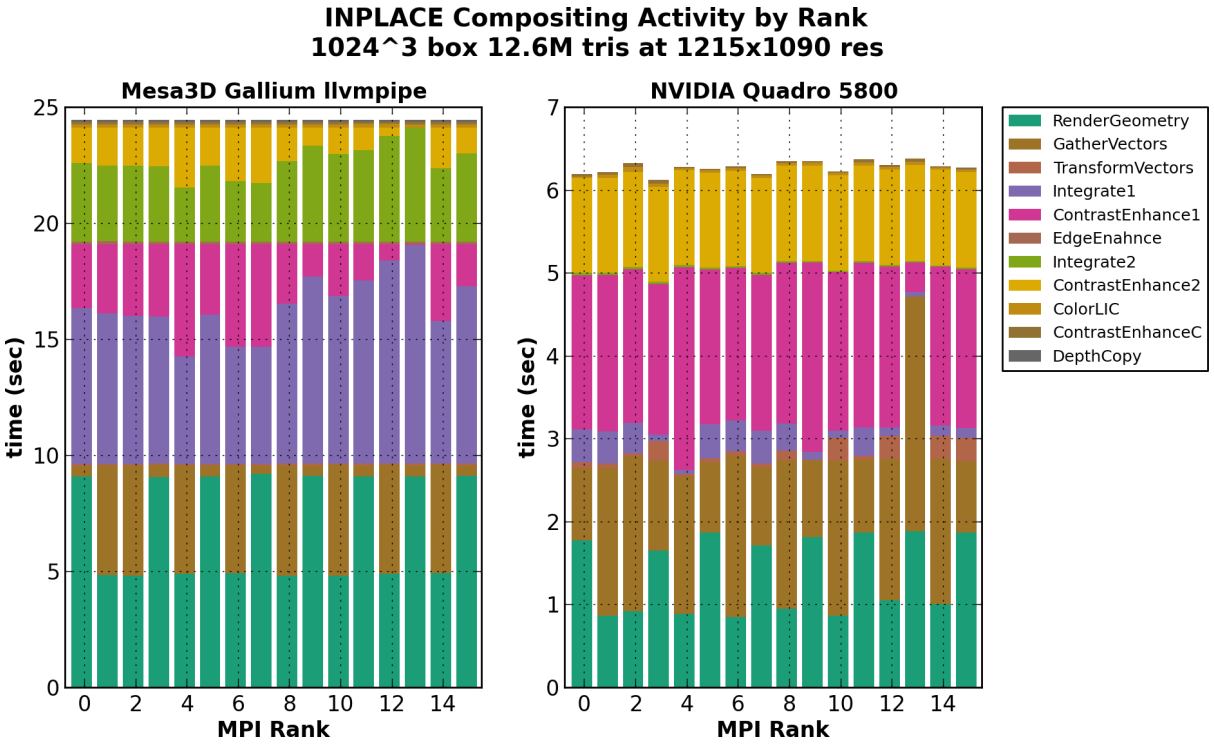


Figure 8: INPLACE Compositing

nearly identical for both compositors. This means that speed up attained by the INPLACE_DISJOINT compositor is a result of reduced integration time while computing the LIC. With INPLACE_DISJOINT approach each rank has fewer fragments to process and there's a greater opportunity to cull empty fragments as the screen space regions are decomposed. These plots also show that the overhead of decomposing(makiing the screenspace disjoint amongst ranks) is small enough to be negligible and that although the INPLACE_DISJOINT moves less data around during compositing and guard pixel exchange it doesn't contribute a lot to the speed of the approach.

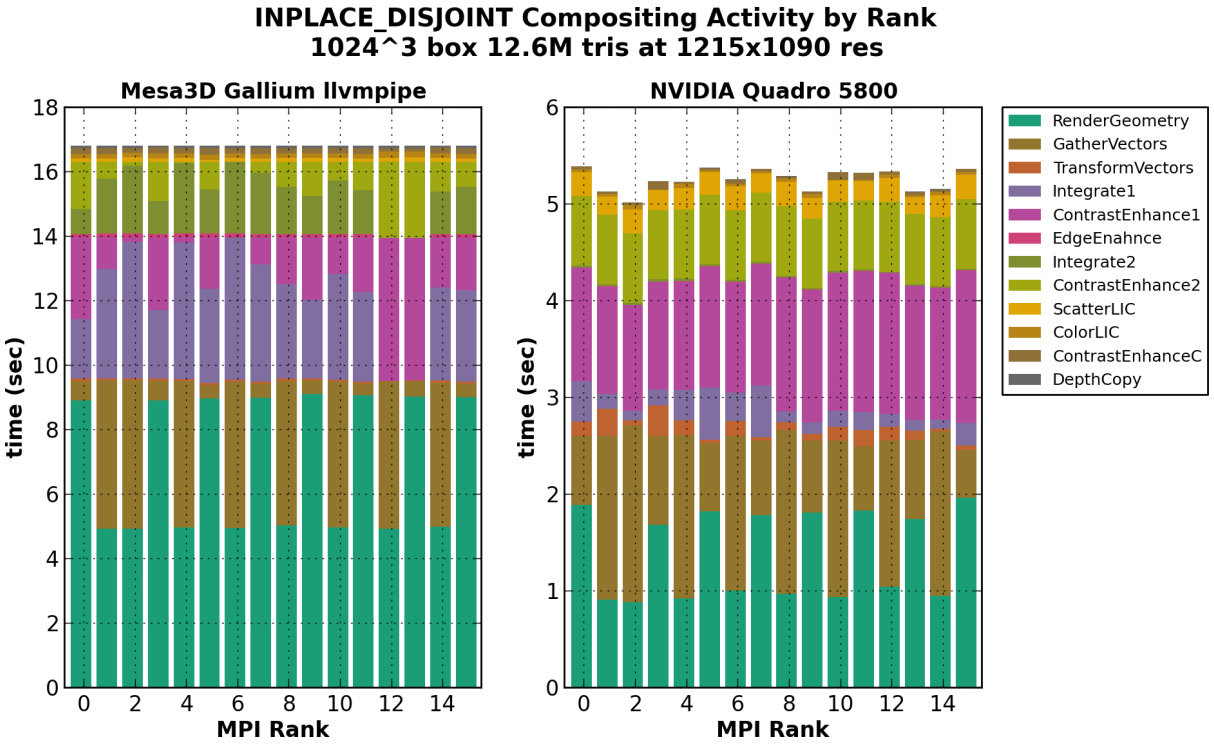


Figure 9: DISJOINT Compositor

5 Apendix A

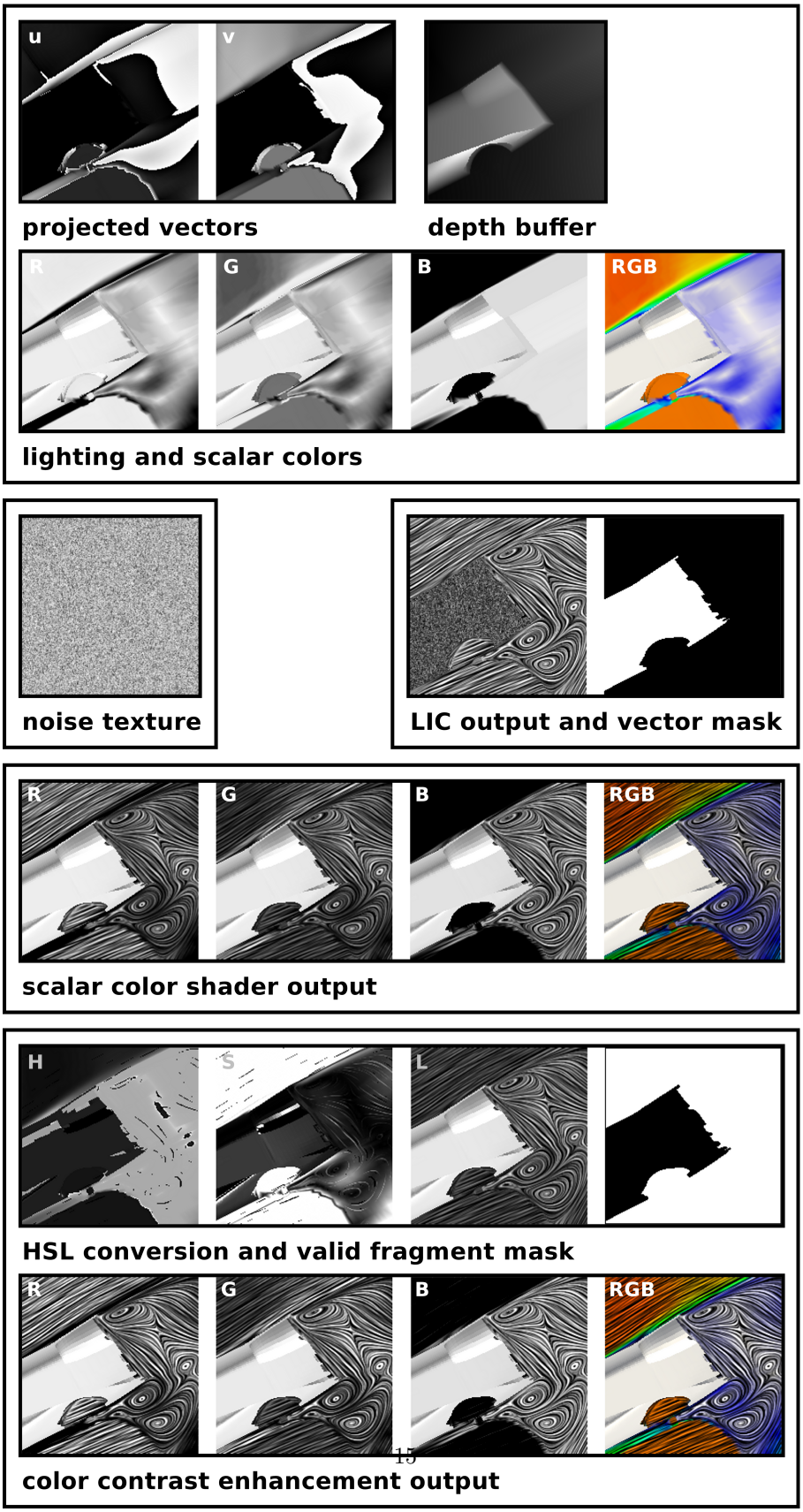


Figure 10: Outputs from the logical processing stages of our surface LIC algorithm. See figure 2 for the stages.

6 Appendix B

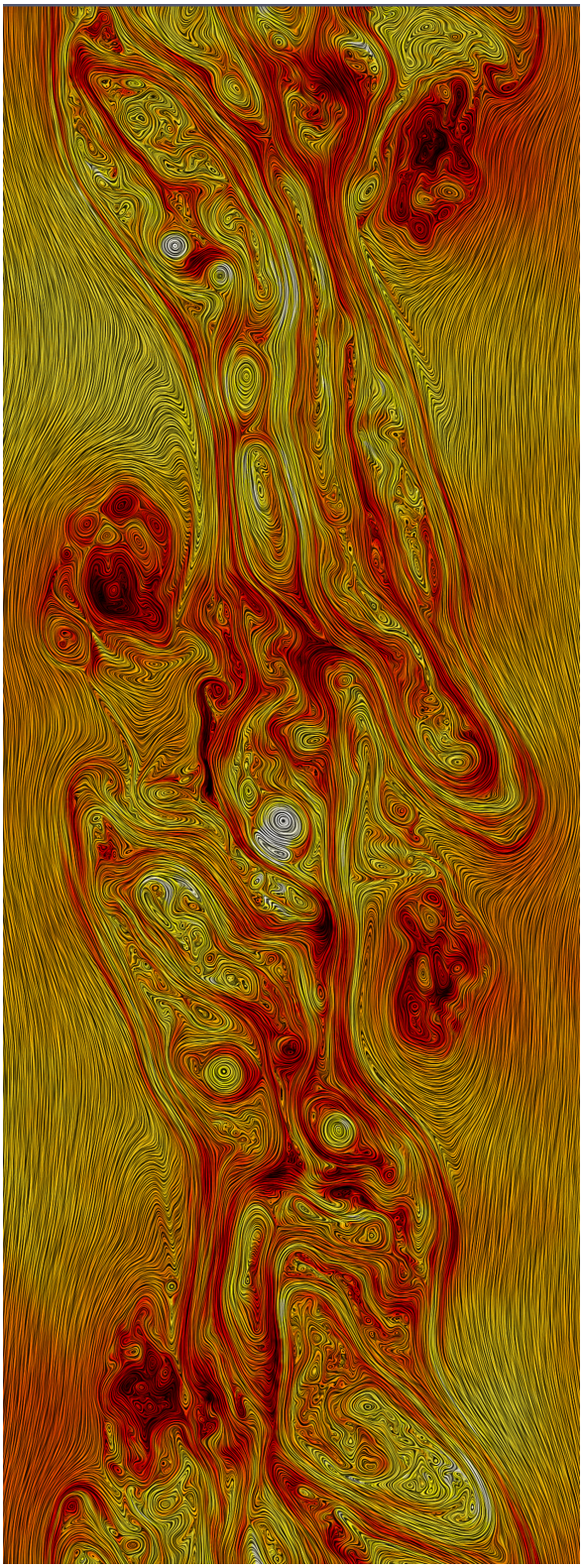


Figure 11: Rendering of the 222M triangle slice dataset used for scaling runs.