Divide and Conquer Spot Noise

Wim de Leeuw
Robert van Liere
Center for Mathematics and Computer Science CWI,
P.O. Box 94097, 1090 GB Amsterdam, Netherlands

Abstract

The design and implementation of an interactive spot noise algorithm is presented. Spot noise is a technique which utilizes texture for the visualization of flow fields. Various design tradeoffs are discussed that allow an optimal implementation on a range of high end graphical workstations.

Two applications are given: the steering of a smog prediction simulation and browsing a very large data set resulting from a direct numerical simulation of turbulence. These applications provide the motivation for the need of interactive visualization techniques.

Keywords: interactive scientific visualization, flow visualization, high performance computing, atmospheric simulation, direct numerical simulation.

1   Introduction.

During the last decade the need for flow visualization techniques for representing vector fields has grown substantially. The reason for this is that increasingly complex phenomena are simulated and that the resulting data sets are becoming more difficult to interpret. The visualization community has developed many accurate and robust flow visualization methods to display these data sets [1]. Simultaneously, there is also a growing demand for interactive computing in which users can control various aspects of the application [2]. The role of computational steering and browsing of very large scientific data bases are two examples of this demand. Both examples require highly interactive visualization to accommodate the requirements for adequate animation and feedback rates.

Spot noise is a texture synthesis technique which can be used to visualize flow fields [3]. Rather than mapping the underlying vector field to produce colored geometric objects, spot noise uses texture. The primary advantage over other flow visualization techniques is that texture can give a continuous view of a 2D field opposed

to visualization at only discrete positions, as with arrow plots or streamlines. The basic idea is that if many particles are used to represent the flow, the individual particles can no longer be discerned and texture is perceived instead. Spot noise generates texture by adding a large number of randomly positioned spots with a random intensity. A spot is defined to be any geometric shape, but usually a small circle is used. Properties of the spot directly control the properties of the texture. Therefore, by modifying the shape of the spot as a function of the data, the data are visualized by texture.

The downside of spot noise is that it is very computationally expensive. A large number of particle paths and particle positions must be calculated, spots must be transformed, scan converted, textured and blended. Previously, spot noise textures were generated as a preprocessing step and the resulting sequence of images were animated. In this paper we discuss the implementation of an interactive spot noise algorithm. Textures are generated on the fly so that spot noise can be used in interactive applications.

Many graphical operations are needed during texture synthesis. Graphics subsystems are designed to handle these operations efficiently. On the other hand, particle position calculations are best computed on general purpose processors. This results in many design software vs. hardware tradeoffs. We discuss these tradeoffs.

The paper is organized as follows: first, the underlying ideas of spot noise are reviewed. We then present and discuss the divide and conquer spot noise algorithm and its implementation. Finally, we apply the algorithm to two applications: the steering of a smog prediction simulation and browsing a very large data set resulting from a direct numerical simulation of turbulence. These applications provide our motivation for an interactive implementation of the spot noise algorithm.

2   Spot noise overview.

We give a brief overview of the spot noise texture synthesis technique. A texture can be characterized by a scalar function f of position $\mathbf{x}$. A spot noise texture [3] is defined as

$$f(\mathbf{x}) = \sum a_i h(\mathbf{x} - \mathbf{x}_i)$$

in which $h(\mathbf{x})$ is called the spot function. It is a function everywhere zero except for an area that is small compared to the texture size. a_i is a random scaling factor with a zero mean, $\mathbf{x}_i$ is a random position. In non-mathematical terms: spots of random intensity are drawn and blended together on random positions on a plane (figure 1).
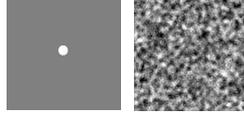
Figure 1: Principle of spot noise: single spot (left) resulting texture (right)

The use of a spot as a basis texture synthesis has a number of convenient, user controllable, properties. First, the shape of a spot determines the characteristics of the texture. By local variation of the spot shape, presentation of data becomes possible; in this way vector fields can be effectively visualized. Second, dynamic phenomena can be displayed via an animated sequence of spot noise images. A spot noise animation of a flow field can be realized by associating a particle with each spot position. A new frame in the animation sequence is determined by advecting all particles over a small distance through the flow field.

Figure 2 is an example of how spot noise can be used to study the separation of a wind field impinging on the front of a block (see section 5.2 for details). The
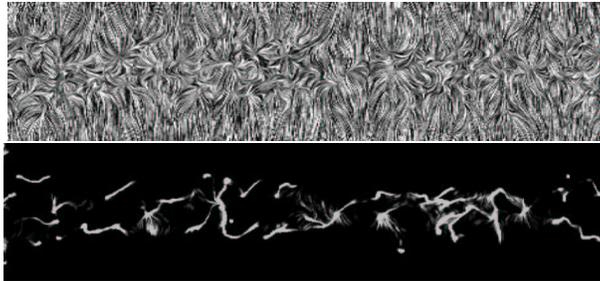

Figure 2: Two views of the separation line of a wind field impinging on a block; default spot noise (top) using advected spot positions (bottom).

question was to find regions where the flow passes over or under the block respectively. The top image shows the skin friction field on the block using the spot noise algorithm with default parameters. By adjusting parameters related to spot position and spot life cycle, the lower image is generated. Adjusting parameters in this way provides the user with a mechanism to highlight certain aspects of the flow which may otherwise stay hidden.

In [4], enhancements to the original spot noise algorithm are given. Bent spots allow spot noise to be used in highly irregular flows, such as in areas where curvature of the vector field is high. Instead of using a single textured polygon with four vertices, bent spots use a textured mesh to represent the spot. The mesh is generated by tiling a surface generated by advecting a stream line in the flow. This is a computationally demanding technique because stream lines must be generated and the mesh must be rendered for each spot. Other enhancements include spot filtering and use of spot noise for non-uniform data grids.

Spot noise pipeline. Algorithmically, spot noise synthesis performs four steps (figure 3):

1. Read a data set of a vector field. For interactive scientific visualization, this step may typically occur anywhere between 5 and 15 times a second.

2. Particle advection. The position of each particle is updated by advecting it by the flow field.

3. Texture synthesis. A spot is generated for each particle position. Each spot is transformed and assigned an intensity according to properties of the flow field. All spots are scan converted and blended together to form a texture map, after which additional spot filtering operations may be applied to the map.

4. An image is rendered by mapping the texture onto a geometric surface. Other visualization techniques may also be superimposed on the texture mapped object.
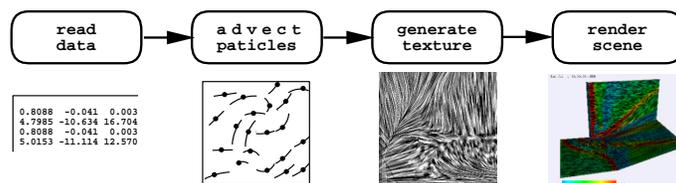

Figure 3: Spot noise pipeline; logical steps (top) and iconic representations (bottom).

Utilizing graphics hardware.   The synthesis of spot noise texture can be considered as rendering and blending of a large number of textured polygons. This functionality is provided by dedicated graphics hardware on most modern workstations. In [4], it was shown how graphics hardware can be exploited to speed up the texture synthesis. The motivation for using graphics hardware is based on the high performance that can be achieved for spot rendering and the higher bandwidth available when manipulating textures. Considerable speedups can be realized when graphics hardware is applied to the last two steps of the pipeline.

In the following sections we use a simplified model of a graphics workstation, as shown in figure 4, to discuss design tradeoffs when implementing the spot noise algorithm. A workstation consists of a number of general processors connected via a bus to the graphics subsystem. Processors have direct access to local caches and memories. The graphics subsystem consists of one or more graphics pipes with very high speed data paths to texture and frame buffers. The graphics subsystem is viewed as a coprocessor, in which the tasks executed on the processors can be performed concurrently with the tasks executed on the graphics subsystem. Each graphics pipe is viewed as an OpenGL state machine which can be set and queried through the OpenGL API and its extensions [5]. The OpenGL API provides access to a rich set of graphics capabilities, including geometry transformations, lighting, texturing, hidden surface removal, blending, and pixel processing.
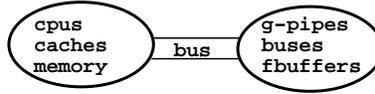

Figure 4: A simplified model of a graphics workstation.

The total time spent to generate N spots is approximated by the equation :

$$T = \max(\sum_{i=1}^{N} genP_i, \sum_{i=1}^{N} genT_i) \qquad (1)$$

in which :

$$genP_i = \text{processor time to calculate position and shape of spot } i$$
$$genT_i = \text{time to blend spot } i \text{ into the texture}$$

The blending step (which is done by the graphics subsystem) overlaps with the position and shape calculation step (done on the processor). The step which takes longest determines the total generation time.

Equation 1 is based on two assumptions. First, in order to move the raw geometric data, there is sufficient bandwidth between the processors and graphics subsystem. Second, in order to avoid under utilization, the data generated by the processors can be streamed to the graphics subsystem. In the following sections we show that both assumptions can be met.

3 Divide and conquer spot noise.

The primary design goal of the divide and conquer spot noise algorithm is to minimize the time needed for texture synthesis. We also identify three secondary design goals: First, to use the capabilities provided by the graphics subsystem; Second, to benefit from multiple processors and multiple graphics pipes; Thirdly, to utilize the high bandwidth in the graphics subsystem effectively.

The basic idea of the divide and conquer method is based on the observations that spots are independent and that the amount of work to be performed on a spot is constant. This allowed us to develop a parallel algorithm in which the work is partitioned evenly between processors and graphics pipes.

The divide and conquer spot noise pipeline is illustrated in figure 5. The collection of particles is partitioned into a number of disjunct sets. Each particle set is processed by one or more processors and exactly one graphics pipe. The processing of a particle set results in an texture. After completion, these textures are gathered and blended to form the final spot noise texture.
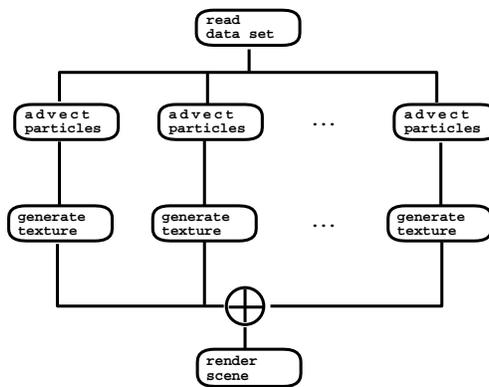

Figure 5: Divide and conquer spot noise pipeline.

Ideally, the total time spent by the divide and conquer algorithm modifies equa-

tion 1 into:

$$T = \max(\sum_{i=1}^{N} genP_i/nP\,,\ \sum_{i=1}^{N} genT_i/nG) + c \qquad (2)$$

in which :

nP	=	number of processors
nG	=	number of graphics pipes
c	=	overhead costs due to additional texture blending

Careful consideration is needed when mapping the divide and conquer pipeline onto an underlying machine architecture. The following, configuration dependent, tradeoffs should be taken into account :

- **balanced resource allocation.** There is a relation between the number of processors and the performance of the graphics pipe assigned to a particle set. This relation can be deduced from equation 2: T will approach a minimum if and only if both nP and nG increase. For example, assigning too many processors to generate particle positions may well saturate the graphics pipe. Alternatively, assigning too few processors will cause starvation, since the processors cannot produce sufficient spots.

- **vertex and texture movement.** Sufficient bandwidth from processors to the graphics subsystem (for vertices) and bandwidth within the graphics subsystem (for textures) is instrumental if optimal performance is to be realized. The number of vertices needed to generate one texture is proportional to the number of vertices per spot times the number of spots. Standard spots consist of four vertices, bent spots are typically defined by about 60 vertices. After each graphics pipe generates a part of the texture, these parts must be combined into the final texture. The final texture size is usually set to 512x512 pixels.

- **OpenGL state machine overhead.** The overhead of setting the OpenGL state machine may be quite substantial. Setting OpenGL in a new state may result in synchronization latencies within the graphics pipe. [1] The tradeoff here is the overhead involved in setting the OpenGL state machine vs. the performance gain of the graphics pipe.

---

[1] For example, SGI's InfiniteReality has four geometry processors which must be synchronized each time a transformation matrix is set.

- **texture decomposition.** Particle sets can be partitioned into disjunct regions, allowing the texture to be decomposed into smaller texture tiles. Tiles, however, are not completely disjunct but will have overlapping boundaries. At the end of the generate texture step the tiles will be blended together to form the final texture. The tradeoff here is the amount of texture space vs. the additional work to be done when blending the final texture.

4   Implementation.

We have implemented the algorithm on a SGI Onyx$_2$ with 8 R10000 processors and 4 InfiniteReality graphics pipes. [2] A 800 MByte/sec bus connects the processors with the graphics pipes.

Since each InfiniteReality is a high performance and capability rich graphics pipe, we have utilized many of its features. An exception to this was the spot transformation which is performed in software by the processors, thus avoiding the high synchronization overhead costs for setting transformation matrices for each rendered spot.

The available processors are partitioned evenly over the number of graphics pipes, as is illustrated in figure 5. For each pipe a process group, consisting of one master process and zero or more slave processes, is created and assigned to the available processors. The task of a master is threefold: it sets up an unique OpenGL graphics context, it renders each calculated spot, and it distributes work among its slaves. The task of each slave is to perform spot shape calculation. If a master process is idle or if there are no slaves (i.e. the number of processors is equal to the number of pipes), the master will also perform spot shape calculation.

We have also implemented texture tiling. Each process group will work only on a predefined region of the final texture. In a preprocessing step, spots are distributed based on location and assigned to the process group dealing with the corresponding region. Spots, however, have a certain extent and may therefore belong to more than one region. Spots for which this might be the case are assigned to each process group they might affect. The advantage of implementing texture tiling this way is that texture tiles can easily be composed to form the final texture. The disadvantage is that some spots are assigned to more than one process group.

Different architectures may result in different implementations. For example, if the OpenGL state machine overhead was smaller then spot transformation could be performed on the graphics pipe. Alternatively, if processors are sufficiently fast

---

[2]We thank the Academic Computing Services Amsterdam, SARA, for allowing us to use their Onyx$_2$ "Reality Monster".

and the caches are large enough then the texture tiles may be competely generated by the processor, bypassing the graphics subsystem altogether.

5 Applications.

Using the described implementation, two applications were selected for performance measurements: an atmospheric pollution model and a direct numerical simulation of a turbulent flow.

5.1 Atmospheric pollution

In [6], computational steering of an atmospheric pollution model is described. The goal is to monitor the evolution of pollutant concentrations while the user can control emission, meteorological and geographical parameters. The output is shown as an animated sequence of images. In [6] arrow plots were used to display the wind fields, which we have now replaced with spot noise textures. During animation, the user can now very clearly see the evolution of a pollutant and how it relates to the flow of the wind field. This relation was much less obvious when using arrow plots.

Figure 6 is a snapshot of the resulting animation, showing a spot noise representation of the wind field and the pollutant O_3 superimposed on it. A rainbow colormap is used for assigning colors to the pollutant. A map of Europe is also drawn.
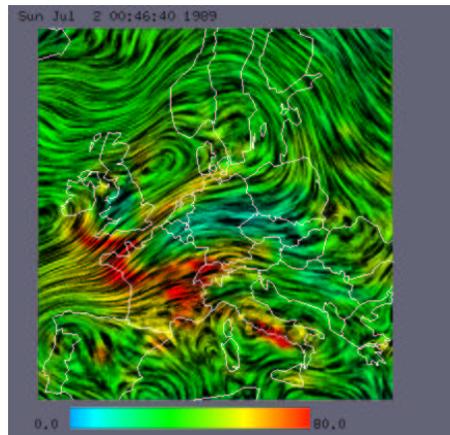

Figure 6: Pollutant O_3 superimposed on the wind field.

The data used is a slice from the three dimensional data set. The data is defined on a regular grid of 53x55 cells. Each texture has a resolution of 512x512 pixels and consists of 2500 spots. Bent spots were used because of the strong fluctuations in the wind field. Each spot is represented as a 32x17 mesh deformed by the flow using stream line integration. The net result is that each texture is generated by approximately 1.3 million quadrilaterals; i.e. 2500x32x17 vertices.

Discussion. Table 1 shows the results of various hardware configurations. The times stated in the table denote the number of spot noise textures that can be generated per second. Only the time for texture synthesis is given; i.e. steps 2 and 3 of the spot noise pipeline. Rows of the table denote 1, 2, 4 and 8 number of processors, respectively. Columns denote 1, 2 and 4 number of graphics pipes.

	1	2	4
1	1.0		
2	2.0	2.0	
4	2.8	3.6	3.9
8	2.7	4.9	5.6

Table 1: Textures per second for atmospheric pollution simulation. Columns denote the number of graphics pipes. Rows denote the number of processors.

The following observations can be made:

- As suggested in equation 2, using more processors does indeed improve the texture generation rate, with a maximum of approximately 4 processors per graphics pipe. Using more than 4 processors per pipe does not increase performance.

- Increasing the number of graphics pipes will also improve the texture generation rate if and only if there are a sufficient number of processors to keep the graphics pipes busy.

- We would expect a near linear performance speedup in the cases that $4n$ processors and n graphics pipes are used. [3] However this is not so, due to the additional overhead caused by blending step (term c equation 2). This blending step is done sequentially.

---

[3]We expect, but have not verified, that when using 4 graphics pipes an optimal performance will be achieved by using 16 processors.

10

- The bandwidth from processor to graphics subsystem is not the limiting factor. At 5.6 textures per second the total bandwidth needed is approximately 116 MBytes/sec for just the raw geometric data, which is well below the maximum of 800 MBytes/sec.

- Using a 32x17 mesh to represent each spot will result in very accurate renderings. Lower resolution meshes will result in less accurate renderings, but can increase performance substantially.

5.2 Direct numerical simulation of a turbulent flow

In [7], methods are discussed for direct numerical simulation of turbulent flow. The goal of this application is to study the evolution of the vortex shedding behind a block, the transition from laminar to turbulent flow, and how these relate to other physical phenomena, such as pressure or helicity. The computation and the size of the resulting data base is impressive: a few weeks of computing can easily produce a few terabytes of data. A data browser is being developed to analyse such scientific data bases. In contrast to prerecorded video sequences, the data browser allows the user to first select visualization mappings and then play through any part of the data base.

Figure 7 is a snapshot of a slice of the data set. The flow is from left to right and impinges a block placed in the field. One can clearly see the transition from laminar to turbulent flow behind the block. Only if the number of frames per second exceeds a threshold can the user monitor how the vortices behave over time.
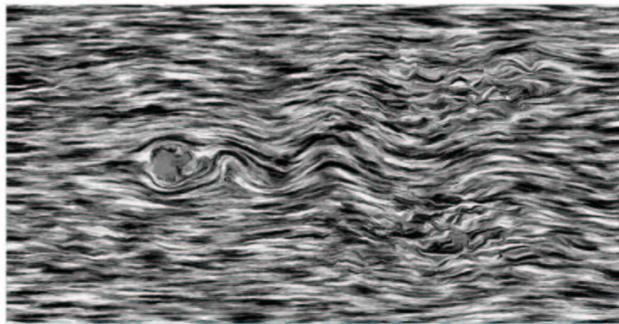

Figure 7: Direct numerical simulation of the wake behind a block showing vortex shedding and transition from laminar to turbulent flow.

The data used is a slice from the three dimensional data set. The data is defined on a rectilinear grid with a resolution of 278x208 cells. Each texture has a resolution of 512x512 pixels and uses 40,000 spots. Bent spots were used because of the turbulent nature of the flow in which strong fluctuations for the flow magnitude and direction occur. Each spot is represented as a mesh with a resolution of 16x3 vertices; i.e. resulting in approximately 1.9 million quadrilaterals per texture.

Discussion. Table 2 shows the results of various hardware configurations. The times stated in the table denote the number of spot noise textures that can be generated per second. The numbers given in table 1 are somewhat higher. Although the mesh resolution of each spot is higher in the atmospheric example, we have used many more spots in this example.

	1	2	4
1	0.7		
2	1.3	1.3	
4	2.1	2.1	2.4
8	2.5	3.2	3.5

Table 2: Textures per second for turbulent flow.

The following observations can be made:

- The structure of table 2 is very similar to that of table 1. The numbers suggest that optimal performance can be achieved if 4 processors are used to drive a graphics pipe. Using more graphics pipes, without increasing the number of processors, will not improve the texture generation rate.

- The amount of data being transported from processors to the graphics pipes is approximately 31.0 megabyte per texture.

- 40,000 spots per texture will result in very accurate renderings. Using less spots will result in less accurate renderings, but can increase performance substantially.

6   Conclusion

In this paper we described how fast generation of spot noise textures can be achieved for the visualization of flow fields. By dividing the collection of spots over a number

of processors and graphics pipes, near interactive speeds can be achieved. Because spot noise allows variation of parameters, speed can be traded for quality and higher speeds than presented in the paper are possible.

Visualization algorithms should be designed to utilize the capabilities of the graphics subsystem. Different software vs. hardware tradeoffs may arise as the performance of processors, memory systems, busses and graphics subsystems increase.

References

[1] L. Hesselink and T. Delmarcelle. Visualization of vector and tensor data sets. In L.J. Rosenblum et al., editors, *Scientific Visualization: Advances and Challenges*, pages 419–433. Academic press, 1994.

[2] L. Tucker and P. Woodward. Visualization methods. In P. Messina and T. Sterling, editors, *System Software and Tools for High Performance Computing Environments*, pages 109–113. SIAM, 1993.

[3] J.J. van Wijk. Spot noise – texture synthesis for data visualization. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 263–272, July 1991.

[4] W.C. de Leeuw and J.J. van Wijk. Enhanced spot noise for vector field visualization. In G.M. Nielson and D. Silver, editors, *Proceedings Visualization '95*, pages 233–239. IEEE Computer Society Press, 1995.

[5] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[6] R. van Liere and J.J. van Wijk. Steering smog prediction. In B. Hertzberger and P. Sloot, editors, *Proceedings HPCN '97*, pages 241–252. Springer-Verlag, 1997.

[7] R.W.C.P. Verstappen and A.E.P. Veldman. Direct numerical simulation at lower costs. *Journal of Mathematical Engineering*, June 1997 (to appear).