Simple Divergence-Free Fields for Artistic Simulation

Mayur Patel[1]
Animal Logic Film

Noah Taylor[2]
Animal Logic Film

Abstract

We discuss tools for calculating divergence-free fields for artistic particle simulations. We introduce a simple, fast divergence-free noise function which can be used for turbulence. We also describe how the interpolation technique used by the noise function can be used for calculating artist-controlled divergence-free fields. The artist can create realistic unbounded flow fields without the complexity or memory cost of voxel-grid methods.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Physically based modeling.

Keywords: simulation, gaseous phenomena, turbulence, physically based modeling, noise, particle systems, incompressible flow

1 Introduction

There are two major frameworks used by computer graphics artists to simulate fluids and gaseous phenomena, particle systems and fluid solvers. Recent advances in fluid solvers have made them computationally feasible for artistic applications [Stam 1999]. Most often, these simulators approximate fluid models, such as the Navier-Stokes equations, using data stored in a regular spatial grid, updating the grid values for each time-step. Particles can later be advected through space using velocities from the grid.

While fluid solvers have made a great deal of simulated effects *possible*, it has not necessarily made the simulation process *easier* for the artist. We believe artists still use fluid solvers selectively, because they often find the particle system easier to control. This is particularly true for choreographed elements where motion needs to be closely directed.

Fluid solvers do provide local-volume effects that are difficult to achieve with the traditional particle system. This is because the fluid solver simulates the medium itself, rather than just the material passing through the medium. One such effect is the divergence-free property of incompressible fluids, that the sum of the flux arriving at a position equals zero. Simulating this effect can cause swirls and eddies to appear plausibly in the flow, adding detail and realism. In this paper, we present fast new interpolation techniques for calculating divergence-free fields. We first interpolate random values to implement a noise function suitable for use as a turbulence field in particle systems. Later, we discuss an interface which allows artists to set desired flows in space. The technique does not require the storage of large grids in memory; data can be stored more compactly in associative data structures. Using this approach, artists can create plausible flow fields that are divergence-free for use in particle systems.

2 Prior Art

There has been a large, on-going collection of work associated with the simulation of fluids and gaseous phenomena for computer graphics. In recent times, fluid solvers have become feasible and a good deal of work has been done to apply them to computer graphics [Foster and Metaxas 1997; Stam 1999; Fedkiw et al. 2001; Nguyen et al. 2002]. Some have reduced computational expense by developing 2-dimensional simulations which are sufficient for specific applications [Witting 1999; Rasmussen et al. 2003]. Several researchers have recognized the difficulty of controlling fluid simulations and have worked to make this process easier [Treuille et al. 2003; Fattal and Lischinski 2004].

There is also a long history of techniques where no simulation is explicitly conducted to produce a flow field. The strategy of combining component flow fields into a composite flow field is widespread. Our technique fits into this family of solutions. Wejchert and Haumann [1991] described techniques for artists to combine component flows, called flow primitives, into a plausible composite flow. Several have focussed on the role of noise components to increase

ACM Journal of Graphics Tools, Vol 10, No 4: 11-22 (December 2005)

1: patelm@acm.org
2: noaht@al.com.au

realism [Ebert and Parent 1990; Stam and Fiume 1993]. In contrast, Weimer and Warren [1999] introduced a subdivision scheme which interpolates plausible flows from sparse specifications. Others have built application-specific systems which are highly optimized for their purpose [Lamorlette and Foster 2002; Holtkamper 2003]. Also, projection techniques are commonly used to condition velocity grids so that they become divergence-free flow fields [Fedkiw et al. 2001; Treuille et al. 2003]. These methods typically use voxel grids so that gradients are easily calculated.

With respect to turbulence, Perlin noise has been used in simulation though it was designed for its visual properties and is not a physically plausible flow field [Perlin 1985]. More recently, Kolmogorov spectrum noise has been used with a good deal of success [Stam and Fiume 1993; Lamorlette and Foster 2002; Rasmussen et al. 2003].

Our new turbulence noise function stores no data, so it consumes less memory than turbulence calculations that require a 2D or 3D grid. Our artist-directed field stores sparse data in an associative data structure, so it is also memory-efficient when compared to grid-based methods. The calculation of data is fast as well, making it feasible to calculate a flow consisting of multiple components fields of different frequencies and amplitudes. This increases the perceived realism and detail.

3 Fast Simulation Noise

Lattice-based noises have enjoyed a great deal of success in computer graphics. They are easy to use and implement, do not require significant amounts of memory, and can be either periodic or non-periodic [Perlin 1985]. Furthermore, they do not require any preliminary computation. They can be calculated for any point in space as needed.

These noises work by dividing the query space into a uniform lattice of cubes. Each corner of the lattice has an associated random value. When a query is performed, the noise function determines into which cube the query position falls, retrieves the values for each corner of the cube, and interpolates them to produce a value for the query position.

Let's call “value noise” the lattice-based noise that linearly interpolates corners that contain random vectors. In this section, we introduce a new noise function, “fast simulation noise.” Instead of interpolating corners, it linearly interpolates flux between faces. In the next section, a variation called “smooth simulation noise” is discussed. Simulation noises superficially resemble lattice-based noises, because they are calculated using a lattice of random values, and they share certain properties as a result. However, the procedure and justification for the simulation noise calculations is unique. We will restrict the following discussion to calculation of noises in three dimensions.

In simulation noise, each edge in the lattice has an associated value. Each face has four component edges, whose values determine the flux at the face. Figure 1 demonstrates how this works.

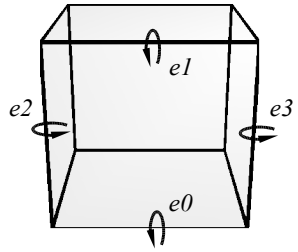

Figure 1: In the front cube face, two edges contribute flux in one direction and two edges contribute flux in the opposite direction. For this particular face, flow into the cube is considered positive. The total flux through the front face of the cube is $f = (e2 – e3) + (e1 – e0)$.

To calculate simulation noise, values are retrieved for each edge of the cube. The flux values for the six faces of the cube are then calculated from the values at edges. Last, the flux between faces is interpolated to give the field value at the query position. Assume that f_{x-} is the flux of the face in the y/z plane with the lower x value, and f_{x+} is the flux of the face with the greater x value. Furthermore, P is the position of interest inside the lattice cube, and P_x is the normalized x component of P, ranging from zero to one in the domain of the lattice cube. If V_{Fx} is the x component of the fast simulation noise field, then its value is calculated with:

$$V_{Fx} = f_{x-} + P_x (f_{x+} - f_{x-})$$

The y and z components can be calculated analogously. Calculating fast simulation noise this way will produce discontinuities in the flow field between lattice cells, whereas value noise would demonstrate C0 continuity.

For a divergence-free field, the Divergence Theorem states that the flux across a closed manifold will total to zero. For simulation noise, this is true. In any cube, each edge contributes flux to two faces. It will contribute incoming flux to

one face and outgoing flux to the other. The total flux for the cube will sum to zero.

A naïve implementation of simulation noise would require 12 edge lookups for each noise calculation. An edge lookup would have a 6-tuple key and return a scalar value. For improved performance, we derive edge values from corner lookup operations. A corner lookup has a 3-tuple key and returns a 3-tuple value. The first random value is associated with the edge extending from the corner in the **+x** direction, the second value with the edge extending toward **+y**, and the last value with the edge extending in the **+z** direction. Only 7 corner lookups are required for each noise calculation. Figure 2 illustrates how corners provide edge data.

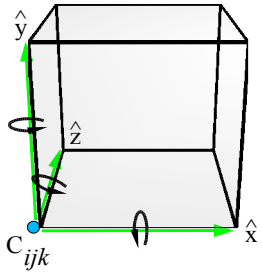

Figure 2: Data for three edges of the lattice are encoded in a 3-tuple associated with a lattice corner, C_{ijk}

For a static field, corner lookups would map to hashing functions. To make simulation noise time-varying, one could map corner lookups to 1D value noise functions, parameterized by time.

Because simulation noise values are interpolated from faces rather than corners, the value for each dimension of the output vector can be calculated independently. As a result, fast simulation noise can be calculated with fewer operations than value noise. Value noise, as we have described it, would require 8 corner lookups, 42 FP add/subtracts, and 21 FP multiplies. Fast simulation noise requires only 7 corner lookups, 24 FP add/subtracts, and 3 FP multiplies.

If V_F is the simulation noise vector calculated at position P, and V_P is the velocity of a particle at position P, then we calculate a force, F, to apply to the particle at P with the following drag-force rule:

$$F = \alpha (V_F - V_P)$$

The value, α, is a coupling coefficient which allows us to loosen the grip that the simulation noise has on the particles. Using small values for α has two benefits. First, fast simulation noise can be used in traditional particle simulations without overpowering the other force fields. Second, discontinuities in the fast simulation noise function are less apparent. The tactic of using a small α works well for some applications, especially if the speed is critical.

4 Smooth Simulation Noise

Fast simulation noise produces discontinuities between adjacent cubes in the lattice. In many applications, discontinuity is not acceptable. There are many interpolation techniques that one could use to produce a continuous field using the same basic framework. Here, we describe one possible solution, called "smooth simulation noise."

Given a continuous step function,

$$step(x) = 6x^5 - 15x^4 + 10x^3 \quad \text{where} \quad 0 \le x \le 1$$

and a continuous bell function which is proportional to the derivative of the step function,

$$bell(x) = 16x^4 - 32x^3 + 16x^2 \quad \text{where} \quad 0 \le x \le 1$$

and a linear interpolation function,

$$mix(a, b, x) = a + (b - a)x \quad \text{where} \quad 0 \le x \le 1$$

we can produce a divergence-free C2-continuous field. The vector components of smooth simulation noise can be calculated from flux values at lattice faces with:

$$V_{Fx} = bell(P_y) \cdot bell(P_z) \cdot mix(f_{x-}, f_{x+}, step(P_x))$$
$$V_{Fy} = bell(P_x) \cdot bell(P_z) \cdot mix(f_{y-}, f_{y+}, step(P_y))$$
$$V_{Fz} = bell(P_x) \cdot bell(P_y) \cdot mix(f_{z-}, f_{z+}, step(P_z))$$

The use of the bell function suppresses flow near lattice edges. Flow is only passed through the center of lattice faces. Since the step function relates to the bell function, it regulates flow between faces such that $\nabla \cdot V_F = 0$.

Smooth simulation noise can be implemented with 7 corner lookups, 36 FP add/subtracts and 30 FP multiplies; its speed is slightly less than linear value noise. The appendix contains an implementation of smooth simulation noise.

A benefit of this interpolation scheme is that no edge values external to a lattice cube need to be retrieved to calculate values within that cube. One drawback is that it has "dead spots" at each lattice edge where the value is zero. Smooth simulation noise is probably best used as a basis function to a fractal with a large number of octaves. The fractal should have an irregular lacunarity value, so that dead spots from one octave are more likely to fall onto live areas of the other octaves.

In figure 3, we demonstrate the differences between value noise and simulation noises by visualizing the output fields of a 2D fractal, using each noise function as a basis. In the case of two-dimensional simulation noises, random scalar values are associated with corners. Flux is calculated for each side by calculating the difference between the values of its two component corners. Sides are then interpolated to produce an output vector.

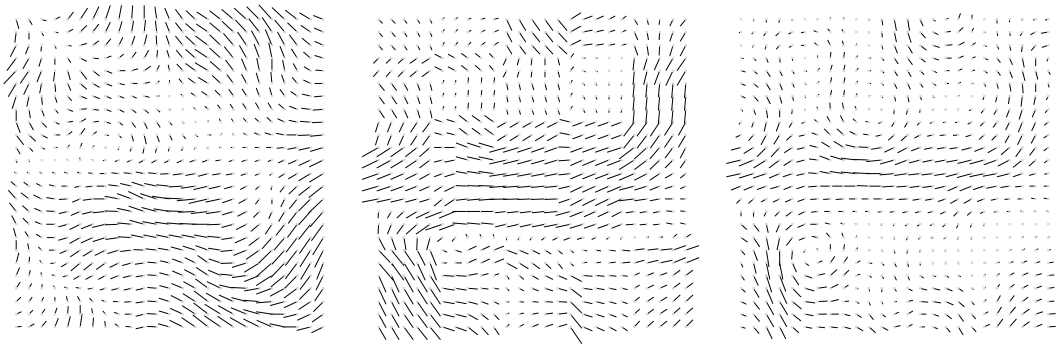

Figure 3: 2D fractal fields using different bases: value noise (left), fast simulation noise (middle), and smooth simulation noise (right). The dimension of the fractal is 0.2, the lacunarity is 1.77 and the number of octaves is 3. Simulation noise shows areas of high flow rate feeding into other areas of high flow rate more realistically than value noise.

The simulation noises we have described only guarantee divergence-free fields. Additional physical effects are not reproduced. Physically-based turbulence functions, such as Kolmogorov spectrum noise, have a higher degree of correctness [Stam and Fiume 1993]. It might be possible to introduce additional physical effects to simulation noises by filtering values returned by corner lookup operations, but this remains to be proven.

5 Artist Control

The interpolation schemes used in simulation noises insure a divergence-free field for any random data associated with lattice corners. The interpolation schemes also work when the data are not random, so this provides an opportunity for artists to edit the field. In a broader sense, the schemes are encoding and reconstruction methods for divergence-free fields, regardless of whether they are random or specified.

User interfaces can be constructed to allow artists to describe the desired flow through space. Solvers can be constructed to take high-level artist descriptions and calculate suitable values to be associated with lattice edges. One obvious approach would be to implement a system of linear equations to solve for edges, given face constraints. We solve the problem with a greedy algorithm rather than analytically. For each face, component edge values are set and locked. If a face has a locked edge, the values are set for the remaining edges. Edge values are later packed into corners and inserted into an associative data structure. In general, it is possible for the user to design spline and lattice configurations that cannot be solved, so implementations need to be fault-tolerant. In our implementation, if no edge of a face is unlocked, then the face constraint is ignored.

In our implementation, artists are able to model directed 3D splines, which are attached to the field as input data. The intersections between the spline paths and faces in the lattice provide flux values for each face. The splines that define flow can be animated with a couple of caveats. First, the lookup data needs to be updated accordingly at each simulation step during which the spline moves. Second, there can be temporal discontinuities as the spline moves from one lattice cube to another. The following figure illustrates an example of a flow defined from a spline.

Figure 4: From left to right, the images show particles moving along a flow derived from one spline. The flow consists of eight component fields of different frequencies and amplitudes superimposed for the final result. The algorithm is sufficiently fast and memory-efficient that the fields are calculated on-line during simulation.

The procedure for calculating the values for the artist-controlled field is very similar to the procedure for calculating simulation noises. The difference is in corner lookup. When calculating the artist-controlled field, corner lookup first consults storage to determine whether the artist has constrained the edge of interest. If so, then the value is retrieved from storage. If not, a random value is calculated using a hashing function, as we do when calculating simulation noise. Flux values for each face can be calculated and interpolated to produce an output vector.

Our user interface uses a fractal noise metaphor with which computer graphics artists are typically familiar. Artists specify a number of octaves to be associated with the field, as well as lacunarity and dimension values. These parameters are sufficient to define multiple lattice spaces, including their frequencies and amplitudes relative to one another. The tool calculates values for the multiple spaces and sums them to produce the total output vector. This allows for increased complexity of the artist-controlled field without a significant increase in interface complexity.

If N is the number of intersections between artist-sculpted splines and faces in the lattice spaces, and if a balanced binary tree is used as the data structure for storing corner data; then the runtime complexity of calculating a point in the field is O(lgN) and the memory complexity is O(N). Caching corner lookups can help to improve performance because consecutive particles often fall within the same lattice cube. The specific value of N is directly under the artist's control. As the artist adds more detailed control, the value of N increases.

We believe that the flow field could be made interactive with the environment by intersecting edges of the lattice with objects in the scene. Because the edges are fixed-length, at fixed intervals, and axis-aligned; we believe that the intersection procedure could be efficient. Any lattice edges that intersect geometry would be constrained to a value of zero. These constraints could be stored for corner lookup just as we did for the artist-directed field. The flow around objects would thereby be suppressed and no flux would pass through scene geometry. This idea has not been tested.

Acknowledgements

Thanks to Animal Logic Film and the *Happy Feet* crew for supporting the development and publication of this technology. Thanks to the reviewers and editors for many helpful comments.

References

Ebert, D. S., and Parent, R. E. 1990. Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques.. In *Proceedings of SIGGRAPH 1990*, ACM Press / ACM SIGGRAPH. 357–366.

Fattal R., and Lischinski, D. 2004. Target-driven Smoke Animation. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, ACM Press / ACM SIGGRAPH. 441–448.

Fedkiw, R., Stam, J., and Jensen, H. W. 2001. Visual Simulation of Smoke. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH. 15–22.

Foster, N., and Metaxas, D. 1997. Modeling the Motion of a Hot, Turbulent Gas.. In *Proceedings of SIGGRAPH 1997*, ACM Press / ACM SIGGRAPH. 181-188.

Holtkamper, T. 2003. Real-time Gaseous Phenomena – A Phenomenological Approach to Interactive Smoke and Steam. In *Proceedings of the 2nd International Conference on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa.* ACM Press. 25-31.

LAMORLETTE, A. AND FOSTER, N. 2002. Structural Modeling of Flames for a Production Environment. In *Proceedings of SIGGRAPH 2002*, ACM Press / ACM SIGGRAPH. 729-735.

NGUYEN D. Q., FEDKIW, R. AND JENSEN, H. W. 2002. Physically-Based Modeling and Animation of Fire. In *Proceedings of SIGGRAPH 2002*. ACM Press / ACM SIGGRAPH. 721-728.

PERLIN, K. 1985. An Image Synthesizer. In *Proceedings of SIGGRAPH 1985*. ACM Press / ACM SIGGRAPH. 287-296.

RASMUSSEN, N., NGUYEN, D. Q, GEIGER, W., AND FEDKIW, R. 2003. Smoke Simulation for Large Scale Phenomena. In *Proceedings of SIGGRAPH 2003*. ACM Press / ACM SIGGRAPH. 703-707.

STAM, J. 1999. Stable Fluids. In *Proceedings of SIGGRAPH 1999*. ACM Press / ACM SIGGRAPH. 121-128.

STAM, J., AND FUIME, E. 1993. Turbulent Wind Fields for Gaseous Phenomena. In *Proceedings of SIGGRAPH 1993*. ACM Press / ACM SIGGRAPH. 369-376

TREUILLE, A., MCNAMARA, A., POPOVIC, Z., AND STAM, J. 2003. Keyframe Control of Smoke Simulations. In *Proceedings of SIGGRAPH 2003*. ACM Press / ACM SIGGRAPH. 716-723.

WEJCHERT, J. AND HAUMANN, D. 1991. Animation Aerodynamics. In *Proceedings of SIGGRAPH 1991*. ACM Press / ACM SIGGRAPH. 19-22.

WEIMER, H. AND WARREN, J. 1999. Subdivision Schemes for Fluid Flow. In *Proceedings of SIGGRAPH 1999*. ACM Press / ACM SIGGRAPH. 111-120.

WITTING, P. 1999. Computational Fluid Dynamics in a Traditional Animation Environment. In *Proceedings of SIGGRAPH 1999*. ACM Press / ACM SIGGRAPH. 129-136.

Appendix

The following pseudo-code implements a smooth simulation noise function.

```
// returns a 3d vector, given a 4d position + time vector
void
simnoise3d( float V[3], const float P[4] )
{
    float       corner[7][3];
    float       face[6];
    float       Pt[3];
    float       b[3];
    signed long Pi[3];
    int         i;

    for(i=0; i<3; ++i) {
        Pi[i] = floor( P[i] );
        Pt[i] = P[i] - float( Pi[i] );
        b[i] = bell( Pt[i] );
    } // end for

    for( i=0; i<7; ++i ) {
        Pi[0] += i & 0x01;
        Pi[1] += (i>>1) & 0x01;
        Pi[2] += (i>>2) & 0x01;

        // this can be a hash, for a 3d input vector;
        // or a 1d noise for 4d position + time input vector
        corner_lookup( &corner[i][0], Pi, P[3] );

        Pi[0] -= i & 0x01;
        Pi[1] -= (i>>1) & 0x01;
        Pi[2] -= (i>>2) & 0x01;
    } // end for

    face[0] = ( corner[4][1] - corner[0][1] ) +
              ( corner[0][2] - corner[2][2] );
    face[1] = ( corner[5][1] - corner[1][1] ) +
              ( corner[1][2] - corner[3][2] );

    face[2] = ( corner[4][0] - corner[0][0] ) +
              ( corner[1][2] - corner[0][2] );
    face[3] = ( corner[6][0] - corner[2][0] ) +
              ( corner[3][2] - corner[2][2] );

    face[4] = ( corner[0][1] - corner[1][1] ) +
```

```
                ( corner[0][0] - corner[2][0] );
    face[5] = ( corner[4][1] - corner[5][1] ) +
                ( corner[4][0] - corner[6][0] );

    V[0] = b[1] * b[2] * mix( face[0], face[1], step(Pt[0]) );
    V[1] = b[0] * b[2] * mix( face[2], face[3], step(Pt[1]) );
    V[2] = b[0] * b[1] * mix( face[4], face[5], step(Pt[2]) );

    return;
}
```