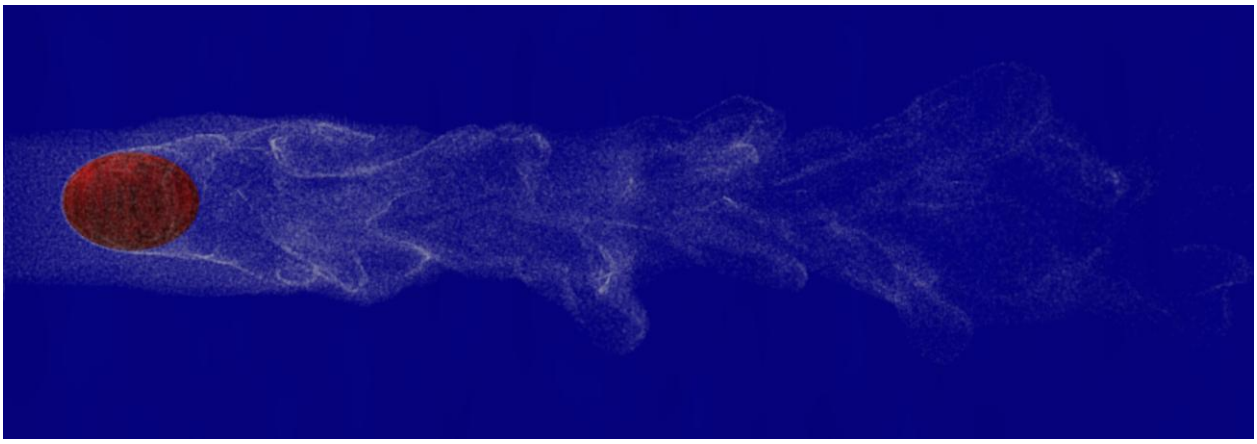
Fluid Simulation for Video Games (part 7)

By Dr. Michael J. Gourlay



Particle Operations for Fluid Motion

This article, the seventh in a series, explains how to integrate a fluid simulation into a typical particle system. Part 1 summarized fluid dynamics; part 2 surveyed fluid simulation techniques; and part 3 and part 4 presented a vortex–particle fluid simulation with two-way fluid–body interactions that runs in real time. Part 5 demonstrated how to profile and use CPU usage data to optimize and further parallelize the code so that it ran faster. Part 6 described how to use a Poisson method to compute velocity from vorticity.

This article introduces features to the simulations presented in previous articles: The fluid flow will include motion resulting from arbitrary external affectors (such as wind and forces), and the fluid simulation will work as particle operations that readily integrate into existing particle systems. These new features enable visual effects authors (including non-programmers) to create compelling, interactive fluid effects using familiar tools.

Particle System Architecture

Visual effects for video games consist of particle systems of screen-based filtering and compositing systems. Fluid simulations naturally fit into particle systems. To understand how they fit, you must first understand some common patterns found in most or all commercial particle systems, such as Houdini, Maya, and Particle Universe, and in McAlllister's Particle System application programming interface (API).

Each effect in a particle system comprises of one or more particle *collections* (sometimes called *groups* or *layers*) and several *particle operations* (sometimes called *actions*). Each operation processes particles in a collection. Operations include emitting, killing, affecting, and rendering. In the code that accompanies this article, the `IParticleOperation` class specifies an interface for these operations:

```
/*! Particle operation abstract base class */
class IParticleOperation
{
    public:
        IParticleOperation() {}
        virtual ~IParticleOperation() {}
        virtual void Operate( float timeStep , unsigned uFrame ) = 0 ;
} ;
```

Each concrete particle operation has a set of parameters that allow effects authors to customize effects. In this implementation, the parameters include a particle collection that the operator acts upon.

Figure 1 shows a schematic of a simplified "fire" particle effect that has "flames" and "smoke" layers. Notice that each layer contains multiple operations; in fact, the list of operations in this example is the same. Parameters (not depicted in the figure) differentiate each layer. So for example, the flames layer might emit at a higher rate, emit particles with a faster rate of rotation, and render each particle with different materials (textures and shaders). And the smoke emitter would be located somewhat above the flames emitter.

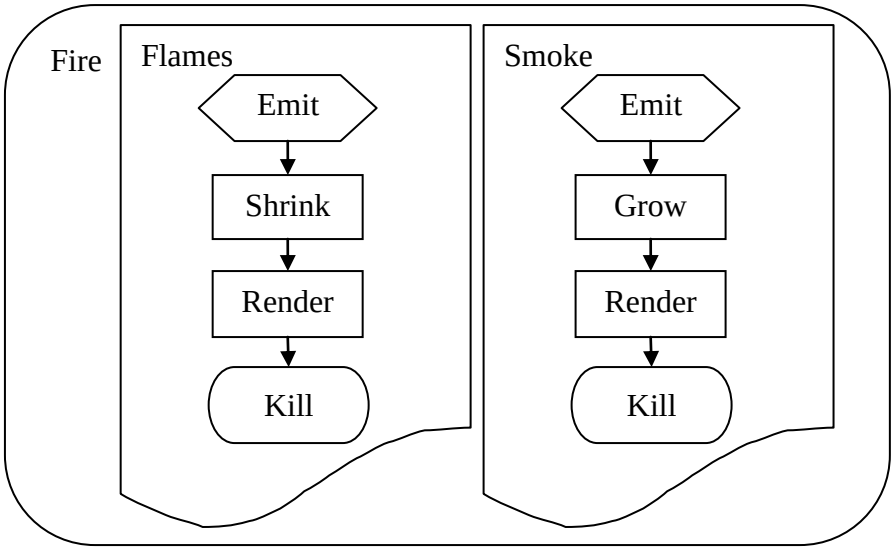


Figure 1. Schematic of a particle effect with multiple groups and operations.

This architecture provides several benefits: Each particle operation has the same interface, so adding custom operations is straightforward. Dataflow and control flow are extremely simple, so once the system has a library of operations, an artist can easily create particle effects without the intervention of a programmer. Furthermore, this data-oriented approach meshes well with the threading paradigm espoused by Intel® Threading Building Blocks (Intel® TBB)—that is, they are easy to scalably parallelize.

The key to making an effective particle system, then, is to make a library of particle operations, such as emitters, killers, affectors, and renderers, that provide maximal power with minimal complexity.

Emission

Most particle effects start with an emitter, which creates particles. The emitter also assigns initial values to each property of a particle. Particle properties typically include:

- **Position.** This 3-vector indicates the location of a particle in space.
- **Velocity.** This 3-vector indicates the direction and speed the particle moves.
- **Orientation.** Commonly, this is a scalar angle that represents the angle to rotate the particle about the axis pointing through the screen. It can also be a 3D value for more sophisticated rendering techniques.
- **Angular velocity.** This is the rate at which the orientation changes over time.
- **Size.** This value indicates the size of the particle—for example, a scalar value, indicating the radius of the particle, s though it were a sphere, or a 2D or 3D value giving more control over the shape of each particle.
- **Mass.** Some affectors apply forces to the particle, in which case the mass of the particle comes into play.
- **Birth time.** This is the (virtual) time at which the particle was created and is used to compute the age of each particle, which in turn can determine other properties, such as any of the above and decide when to kill each particle.

Particles can have additional properties, such as target position and charge, which can be used for more specialized behaviors.

Emitters typically specify a range of initial values for each property, and the actual values for each property are randomly generated to lie within that range. For example, initial position might lie within a region. The region might be a box, cylinder, sphere, or other shape, such as on the surface of a complicated geometric mesh.

Killing

Usually, each particle lives for a finite duration. Criteria for killing particles can be practically anything you can imagine, but commonly they depend on age or position; particles older than a given amount or lying beyond a specified region get killed.

Killing particles serves the practical purpose of keeping particle counts small enough so that they do not consume excessive resources, such as memory and CPU or GPU time.

Forces, Advection and Other Affectors

Particle operations called *affectors* can evolve particle properties over time. For example, a force field operation can apply a constant force, such as gravity or buoyancy, making particles accelerate downward or upward.

The simplest affectors apply the same force or change in motion uniformly, but more sophisticated affectors can apply a different motion at each location in space. An affector that assigns particle velocity based on the particle's position is called an *advection field* (that is, a velocity field), such as the one depicted in Figure 2, which should by now seem familiar; every article in this series—and in fact all of fluid simulation—entails computing an advection field that control the motion of fluid.

In an advection affector, each particle moves according to a field, where velocity depends on location in space.

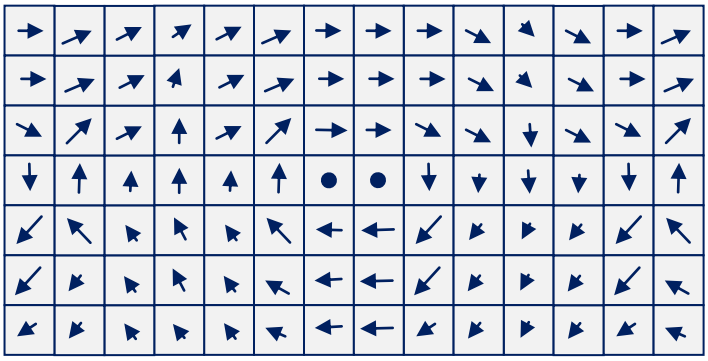


Figure 2. An advection field

Traditionally, advection fields have been considered too computationally expensive for a video game to generate at run time, so they are generated offline in the content-creation and conditioning pipeline, and then imported into the game as another asset, like a texture. The resulting pre-baked motion seems fluid-like but is not interactive; it does not respond to changes in the virtual environment. But these articles present a fluid simulation that runs fast enough to be used at run time.

Rendering

Particles are commonly rendered as camera-facing quadrilaterals called *billboards* or *sprites.* Typically, each particle is rendered with a texture that has variable opacity so that when particles are rendered over each other, they give a passable illusion for occupying a volume. More sophisticated techniques exist; generating surfaces (such as *isosurfaces*) can make a collection of particles appear as a body of liquid. Bahnassi's "mega particles" method uses lit spheres and post-processing.

Fluid Simulation as Particle Operations

The previous articles in this series described a fluid simulation using *vortons*—vortex particles that influence the motion of other particles. Each vorton is a source of circulation, which propels fluid in swirling motions. One stage of that fluid simulation generates an advection field. This article describes advection fields as yet another particle operator. Using the fluid simulation presented in this series within a particle system entails rewriting the simulation code in terms of particle operations. If you treat advection with some care, you can treat fluid motion as just another particle operation and plug it into an existing particle system.

Particle systems are suited to representing vortons, because they are simply particles—vortex particles that have the same properties as other particles (described above in the section, "Emission"). Vorticity advects, like material quantities such as mass and passive tracers—that is, their positions obey the same rules. Furthermore, angular velocity is related to vorticity as $\omega=2\Omega$—that is, vorticity is twice angular velocity, so there is no need for an additional "vorticity" property. So, it is reasonable to reuse regular particles to represent vortons.

The code accompanying this article includes several particle operations. This article omits descriptions of simple canonical operations such as emitting, killing, and rendering. But some operations are peculiar and warrant descriptions:

- **Find bounding box.** Find an axis-aligned bounding box that contains all particles. Doing so helps determine the domain of the advection field, which must encompass all particles that the advection field will influence.
- **Vorton sim.** Given a bounding box, compute an advection field (as a velocity grid), and then diffuse and stretch vorticity.
- **Advect.** Move particles according to a given advection field (as a velocity grid).
- **Fluid–body interaction.** Given a list of rigid bodies, change particle properties so that the fluid satisfies its boundary conditions (no-slip and no-through), and then apply impulses to rigid bodies to conserve momentum and energy.

This approach lets us treat both vortons and passive tracers using the same particle operations, reducing code complexity and yielding some surprising benefits, such as a simple way to combine fluid–body interactions that take into account *any* source of motion, not just vortical fluid motion.

Note: The fact that this simulation happens to use vortex particles to generate the advection field is a mere coincidence to its use within a particle system. The most important thing is that the simulation generates an advection field; how it does so could be different and not use particles. But as described below, the simulation exploits the fact that the source of the advection field is itself a collection of particles to code the interaction elegantly.

Find Bounding Box

The `FindBoundingBox` particle operation finds the axis-aligned bounding box of a set of particles. (Part 3 and part 5 in this series present the details of that algorithm.) It stores the results as the

minimal and maximal corners of the box. These results feed into the `VortonSim` operation. Here is the code:

```
/*! Find bounding box of a dynamic array of particles. */
class PclOpFindBoundingBox : public IParticleOperation
{
    public:
        PclOpFindBoundingBox( void )
            : mParticles( 0 )
            , mMinCorner( FLT_MAX , FLT_MAX , FLT_MAX )
            , mMaxCorner( - mMinCorner )
        {}
        void Operate( float timeStep , unsigned uFrame )
        {
           Particles::FindBoundingBox( * mParticles, mMinCorner, mMaxCorner );
        }
        const Vec3 & GetMinCorner( void ) const { return mMinCorner ; }
        const Vec3 & GetMaxCorner( void ) const { return mMaxCorner ; }
        Vector< Particle > *    mParticles ;
    private:
        Vec3                    mMinCorner ;
        Vec3                    mMaxCorner ;
} ;
```

Vorton Sim

The `VortonSim` particle operation includes a `VortonSim` object, which in turn contains a dynamic array of vortex particles and pointers to the minimal and maximal corners of an externally specified bounding box. The `VortonSim Update` performs these steps:

1. Find the bounding box of the vortons.
2. Aggregate the bounding box (which initially just includes vortons) to include an externally specified domain (which includes the passive tracers).
3. Update the `VortonSim` (as described in part 3).

In the context of a particle system, this operation creates a velocity grid to use in an "advect" particle operation. It also updates vorticity (stored as angular velocity), which changes as a result of diffusion and stretching. But it does *not* update positions or velocities. Subsequent operations do that.

Combining Velocity Fields

Separating the generation of the velocity field from its use allows playing a sneaky trick: adding other motion that the fluid–body interaction respects. Imagine that an effects author wants particles to blow in the wind—with the same direction and speed everywhere—and also wants to add some turbulence and swirls that blow around objects. The fluid simulation can handle the turbulence, but adding a "mean flow" such as a background wind (for example, blowing toward the west at 10 miles per hour) is easier and more familiar to accomplish with a "wind" operation (which every particle system already includes).

Making this trick work requires two parts. The first part entails *adding* the velocity of the advection field to the velocity created by other operations. At the end of each update, each particle knows its total velocity resulting from the combined influence of all affectors.

The second part of the trick entails performing the fluid–body interaction *after* computing the *combined velocity.* The next section explains why.

Fluid–Body Interaction

The `FluidBodyInteraction` operation has these members:

- A pointer to a velocity grid (computed by `VortonSim`)
- A pointer to a dynamic array of rigid bodies (owned by an external physics engine)
- A Boolean value indicating whether to treat particles as tracers or vortons.

  This value indicates whether a particle–body collision should consider the linear or angular velocity of the particle. The former applies linear impulses to rigid bodies, whereas the latter apply torques. Furthermore, this operation changes vorticity to satisfy no-slip and no-through boundary conditions. (See part 4 of this series for details.) This dichotomy between tracers and vortons is somewhat artificial; in principle, the simulation could consist entirely of vortons, without any passive tracers. But to keep computational cost down, the simulation has far fewer vortons than passive tracers. Vortons *must* change their vorticity (which is related to angular momentum) to satisfy boundary conditions. But the simulation has more tracers; hence, using them yields better spatial resolution. The simulation therefore relegates linear momentum to tracers and angular momentum to vortons when applying impulses to rigid bodies.

This operation solves boundary conditions using the algorithm described in part 4 but with one simple but very useful change: The former algorithm obtained flow velocity from the velocity grid. That sufficed when the velocity grid contained all the information about vorton velocity. But the new scheme allows other operations to add velocity to the vortons. So, to account for that "extra" velocity, the `SolveBoundaryConditions` algorithm uses the vorton velocity itself to approximate the fluid velocity at the fluid–body interface. This trick also happens to run faster, because it skips the interpolation of the velocity grid.

This approximation is somewhat inaccurate, because the vorton lies some distance away from the actual fluid–body boundary. This approach therefore lacks mathematical rigor. But in practice, it yields passable results adequate for visual effects in video games; fluid moving past rigid bodies induces vorticity with the right sense. Vorticity sheds in more or less the same way you would expect.

Note: As mentioned in part 4, this ad hoc treatment of fluid–body interactions and boundary conditions makes the drastically simplifying assumption that the velocity contribution from the vorton closest to a point dominates the flow velocity there. Because the velocity induced by a vorton decays with distance squared, that assumption might naively seem reasonable, but it neglects the influence of a large number of *coherent* vortons, such as large vortex tube or

sheet. A proper treatment of boundary conditions requires inclusion of non-local influences. See "Further Reading" in part 4, for more information.

Advect

The `Advect` particle operation has a pointer to a velocity grid that represents the advection field, computed by `VortonSim`. It performs the interpolation and *adds* (not overwrites) velocity to each particle.

Putting It All Together

The fluid simulation particle system has this form, as coded in the `InteSiVis` constructor:

1. Kill old vortons.
2. Kill old tracers.
3. Emit vortons.
4. Emit tracers.
5. **Find bounding box of tracers.**
6. **Update vortons vorticity, and compute advection field as a velocity grid.**
7. Apply wind and other affectors to vortons.
8. Apply wind and other affectors to tracers.
9. **Advect vortons using the velocity grid.**
10. **Advect tracers using the velocity grid.**
11. **Perform fluid–body interaction on vortons.**
12. **Perform fluid–body interaction on tracers.**

Bold items are peculiar to this fluid simulation. Other items are simply canonical particle operations that any existing commercial particle system already supports.

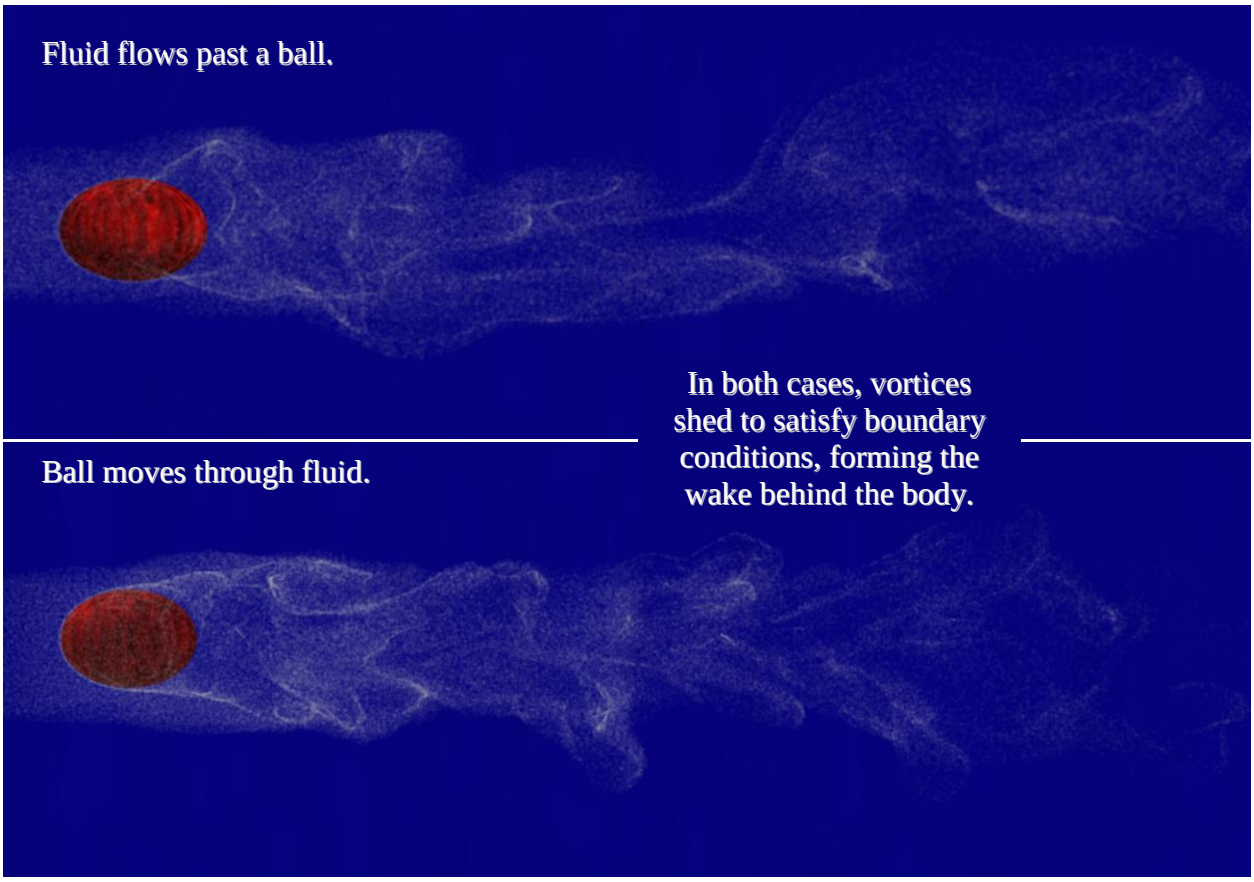


Figure 3. Fluid-body interactions. The upper image shows fluid motion that includes both mean flow (from a wind particle operation) and vortices. The lower image shows a ball moving through fluid that has no externally imposed mean flow; its motion comes entirely from vortices.

In these example effects, bodies only feel the influence of the fluid where both vortex and tracer particles exist, but the bodies can move outside regions where those particles reside (see Figure 3). A better solution would entail emitting particles all around the body and connecting an emitter to each rigid body. Feel free to experiment.

Integrating the fluid simulation into an existing particle system adds the ability of fluid flow to include motion resulting from external affectors (like wind) and enables visual effects authors to create intricate, responsive fluid effects using familiar tools.

Now go add interactive eye candy to your games, and let us know where we can go play in your virtual worlds!

Further Reading

- Particle Systems – a Technique for Modeling a Class of Fuzzy Objects.