

Fluid Simulation for Video Games (part 15)

By Dr. Michael J. Gourlay

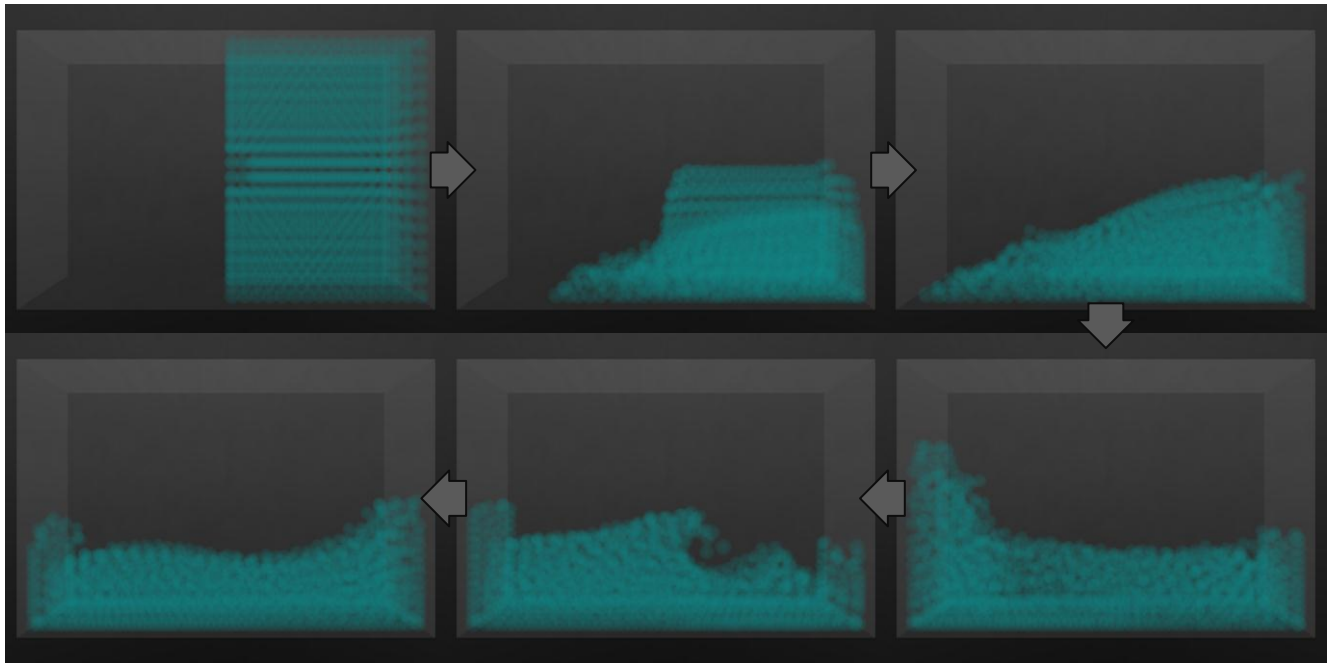


Figure 1. A fluid column falls, sloshes to the left, forms a peak that travels right, and breaks.

Smoothed Particle Hydrodynamics

Fluids come in a wide variety of forms, such as gas, liquid, and goo. The vortex particle method described in previous articles does a good job of simulating gasses and fine filamentary movements. But the simulation offered in [part 14](#) had trouble dealing with containers. And to handle liquids, it's natural to put them in a container.

Smoothed Particle Hydrodynamics (SPH) is a fluid simulation technique. Like vortex particle methods, SPH uses particles to represent parcels of fluid, but SPH directly solves the momentum equation, whereas vortex particle methods solve the vorticity equation. In the vernacular introduced in [part 1](#) and [part 2](#), SPH uses a velocity-pressure formulation, whereas vortex particle methods use a velocity-vorticity formulation. In contrast to vortex particle methods, each SPH particle has only a local influence, which makes it conceptually easier to handle colliding with the boundaries of a container.

This article—the 15th in the series—describes a rudimentary SPH fluid simulation used to model a fluid in a container. [Part 1](#) summarized fluid dynamics; [part 2](#) surveyed fluid simulation techniques. [Part 3](#) and [part 4](#) presented a vortex-particle fluid simulation with two-way fluid-body interactions that run in real time. [Part 5](#) profiled and optimized that simulation code. [Part 6](#) described a differential method for computing velocity from vorticity, and [part 7](#) showed how to integrate a fluid simulation into a typical particle system. [Part 8](#) explained how a vortex-based fluid simulation handles variable density in a fluid; [part 9](#) described how to approximate buoyant and gravitational forces on a body immersed in a fluid with varying density. [Part 10](#) described how density varies with temperature, how heat transfers throughout a fluid, and how heat transfers between bodies and fluid. [Part 11](#) added *combustion*, a chemical reaction that generates heat. [Part 12](#) explained how improper sampling caused unwanted jerky motion and described how to mitigate it. [Part 13](#) added convex polytopes and lift-like forces. [Part 14](#) modified those polytopes to represent rudimentary containers.

Discretization

As [part 2](#) described, every fluid simulation algorithm begins with *discretization*—breaking up the continuous fluid into chunks so that a computer can operate them. Underlying SPH is a general way to interpolate values of functions and their derivatives given a cloud of points.

Representing Functions with Particles

Say you want to know the value of some function A at some point \vec{r} . You don't know the function at all points in space, but you know it at *some* points in space, and you have no control over where those points lie. In other words, your function is randomly sampled. This sounds almost like a problem that the [Monte Carlo method](#) solves: Given random samples of a function, compute its integral. The trick then becomes how to use an integral of a function to evaluate that function.

Monte Carlo Integration

Say you want to compute this integral η :

$$\eta(\vec{r}) = \int_V w(\vec{r} - \vec{r}') A(\vec{r}') \rho(\vec{r}') dV'$$

Say you only know the values of A at N random points \vec{r}_i . Also, $\rho(\vec{r}') dV'$ is the probability of a point being inside the volume element. [Monte Carlo integration](#) approximates the integral with a sum over those points:

$$\eta(\vec{r}) \approx \frac{1}{N} \sum_{i=1}^N w(\vec{r} - \vec{r}_i) A(\vec{r}_i)$$

Function Values from Integrals

Suppose w is a [Dirac delta function](#) δ . When used as an [integral kernel](#), δ “picks out” the value of the integrand at a single point. The integral $\eta(\vec{r})$ just becomes the value of the integrand $A(\vec{r}')\rho(\vec{r}')$ at the query point \vec{r} : $A(\vec{r})\rho(\vec{r}) = \int_V \delta(\vec{r} - \vec{r}')A(\vec{r}')\rho(\vec{r}')dV'$. This gives the value of a function from an integral.

By itself, this does not solve the problem; if you already knew the value of a function at the query point, you would not need to interpolate it.

You can compose the Dirac delta function by choosing a spiky function and simultaneously growing its height and shrinking its width, taking the limit as the spike becomes infinitely tall and infinitesimally narrow. You could stop short of taking that limit all the way and leave the kernel spread out. Plug that smooth kernel into the Monte Carlo sum above as w . Viola! Now you can approximate the value of a function from a sum over randomly placed points (as in Figure 2).

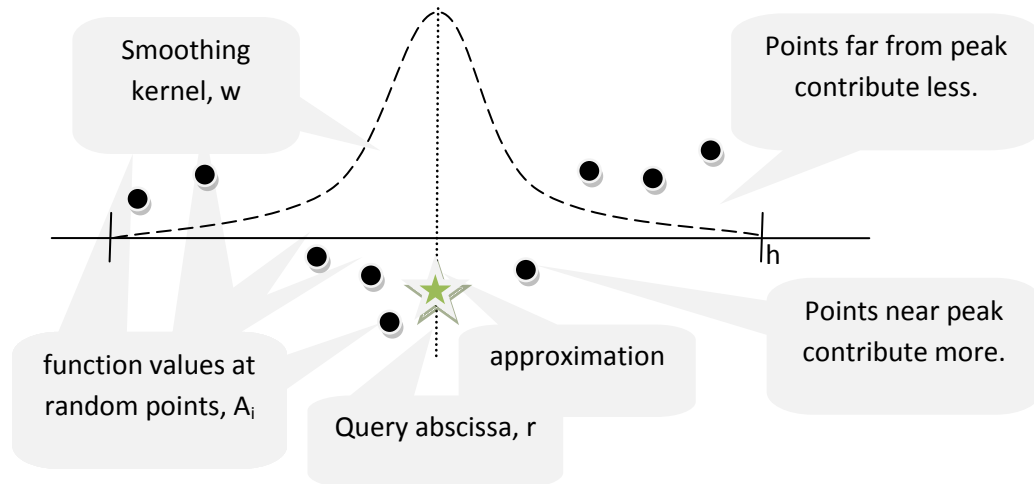


Figure 2. Using a weighted sum to approximate a function at a query point

Monte Carlo Meets Particles

To use this trick, recognize that these randomly placed sample points coincide with particles. The number density at each particle is ρ_i , and the value of some quantity A at each particle is A_i . You can write the summation above as:

$$A_s(\vec{r}) \approx \sum_{i=1}^N \frac{A_i}{\rho_i} w(\vec{r} - \vec{r}_i)$$

Note: Most treatments of SPH establish ρ as the mass density, in which case the formula above has an extra factor: the particle mass.

Smoothing Kernels

You have a choice of kernel, as long as it has certain properties:

- ❑ It should satisfy the unity condition, $\int w(\vec{r})d\vec{r} = 1$.
- ❑ Higher-order moments should be zero.
- ❑ To be computationally expedient, the kernel should have [compact support](#), meaning it is zero beyond some finite range—that is, $w(\vec{r}) = 0$ for $r > h$.
- ❑ It should be non-negative.
- ❑ It should be smooth and differentiable. (The next section shows why.)
- ❑ It should avoid clustering (explained below).

Table 1 and Figure 3 show some kernel forms, where $q \equiv r/h$ and each algebraic function is truncated to be zero for $r > h$. (For brevity, the table omits the normalization factor, which depends on dimension, required to satisfy the unity condition.)

Table 1. Smoothing kernel forms

Name	Kernel form	Derivative	Citation
Gaussian	$\exp(-4q^2)$	$-8q \exp(-4q^2)$	Gingold & Monaghan (1977)
	$(1 + 3q)(1 - q)^3$	$-12q(1 - q)^2$	Lucy (1977)
Spiky	$(1 - q)^3$	$-3(1 - q)^2$	Hoover, et al. (1994)
Poly6	$(1 - q^2)^3$	$-6q(1 - q^2)^2$	Müller, Charypar, & Gross (2003)

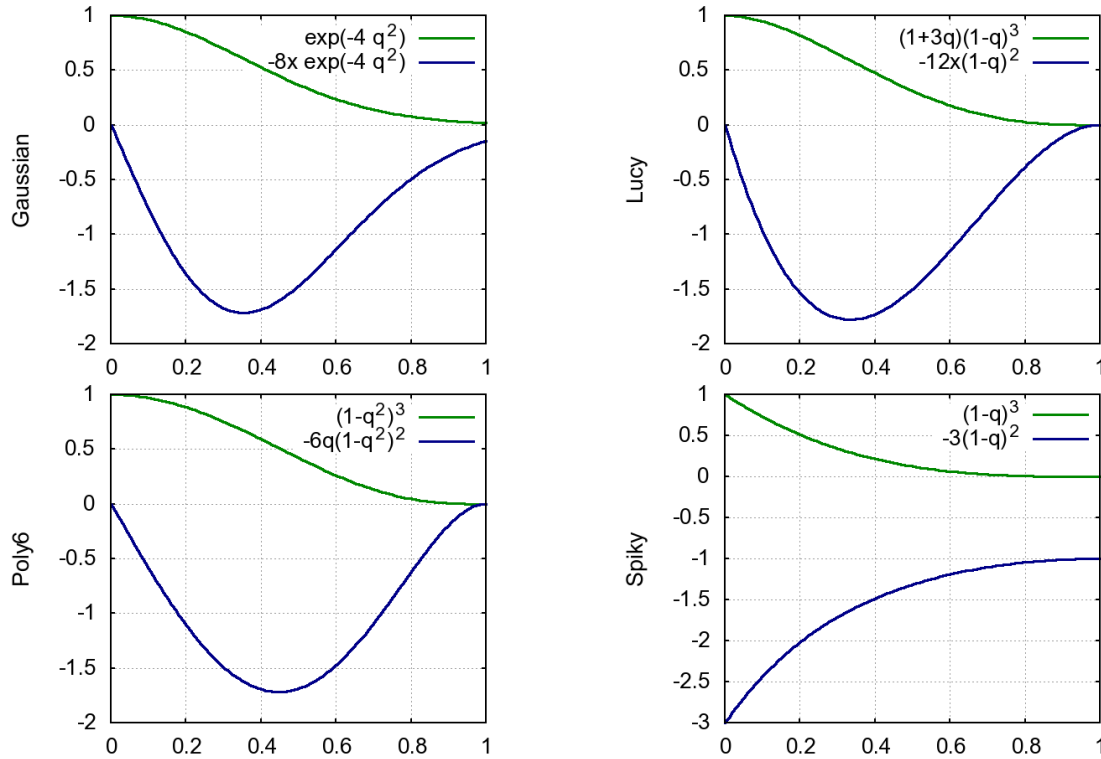


Figure 3. Smoothing kernels and their first derivatives

The best choice for h depends on the number of neighboring particles, but for the sake of simplicity, the sample code accompanying this article uses a uniform value.

Spatial Derivatives

Fluid simulation entails solving partial differential equations. You must compute not only functions but also their spatial derivatives.

SPH can represent spatial derivatives using a variety of formulae. Table 2 shows some forms. All use the gradient of the smoothing kernel. (That's why the kernel should be differentiable.) The most straightforward (canonical) uses only that gradient and the values of the functions at each particle.

You can derive other formulae by taking the gradient of a product of functions:

$$\nabla(A\rho) = \rho\nabla A + A\nabla\rho$$

Rearranging terms leads to the difference formula in Table 2:

$$\rho\nabla A = A\nabla\rho - \nabla(A\rho)$$

$$\nabla A = [A\nabla\rho - \nabla(A\rho)]/\rho$$

The difference formula has an advantage in some cases. Points near boundaries lack neighbors in some directions, so the sum has no contributions in that direction. That causes the gradient at boundaries to point inward toward where neighbors lie. If the field continues even where no points lie, then the difference formula better approximates derivatives of that field.

Note: What does this imply for the gradient-at-walls problem? The difference model avoids spurious density gradient at walls but misses desirable gradients (e.g., at the edges of droplets or bubbles). Ponder this in the context of [part 14](#).

Taking the gradient of a quotient leads to the symmetric formula in Table 2:

$$\nabla\left(\frac{A}{\rho}\right) = \frac{\rho\nabla A - A\nabla\rho}{\rho^2}$$

$$\frac{\nabla A}{\rho} = \frac{A\nabla\rho}{\rho^2} + \nabla\left(\frac{A}{\rho}\right)$$

The symmetric form facilitates obeying Newton’s third law; the force a first body exerts on the second is equal in magnitude and opposite in direction of the force the second exerts on the first. The canonical formula lacks this property, and that leads to momentum not being conserved.

Table 2. SPH gradient formulae

Canonical	$\nabla A(\vec{r}) = \sum_i \frac{A_i}{\rho_i} \nabla w(\vec{r} - \vec{r}_i)$
Difference	$\nabla A(\vec{r}_j) = \frac{1}{\rho_j} \sum_i (A_i - A_j) \nabla w(\vec{r}_{ij})$
Symmetric	$\nabla A(\vec{r}_j) = \rho_j \sum_i \left(\frac{A_i}{\rho_i^2} + \frac{A_j}{\rho_j^2} \right) \nabla w(\vec{r}_{ij})$

In these formulae, the subscript j means “the value for the particle at \vec{r}_j ,” and \vec{r}_{ij} is the separation between two particles, $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$.

Note: See Colin, Egli, & Lin (2006) for visualizations and analysis of these gradient formulae.

Physical Model

Recall from [part 1](#) that the Navier-Stokes equations describe fluid motion. The momentum equation governs how particles move as a result of inertial, internal, and external forces:

$$\frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \nabla) \vec{v} = \frac{-\nabla p}{\rho_m} + \frac{\mu}{\rho_m} \nabla^2 \vec{v} + \frac{\vec{f}_{ext}}{\rho_m}$$

Note that ρ_m refers to mass density, whereas in the preceding sections, ρ stood for number density. They're related: $\rho = \rho_m/m$, where m is the mass of the particle.

Simulating a fluid using SPH entails solving the momentum equation; local density deviations drive pressure gradients, which accelerate particles. (In contrast, vortex methods advect fluid without directly using pressure.)

An SPH simulation involves repeating these steps as time advances:

1. Compute density at each particle.
2. Compute pressure from density.
3. Compute forces from pressure gradients.
4. Compute external forces.
5. Apply those forces to move particles.

Density

To compute number density at each particle, employ the SPH formula:

$$\rho_s(\vec{r}_j) \approx \sum_{i=1}^N w(\vec{r}_{ij})$$

Pressure

Pressure is related to density via a thermodynamic [equation of state](#). To model a gas, use the ideal gas law, where pressure is proportional to density. SPH simulations can use a pseudo-pressure that is proportional to number density:

$$p_i = k(\rho_i - \rho_0)$$

where p_i is the pressure at particle i , ρ_i is its density, ρ_0 is a target density, and k is a stiffness parameter (like a spring constant) whose value is related to the propagation speed of pressure waves. (Effectively, $k = c_s^2$, where c_s is the speed of sound.) This makes particles behave somewhat like a spring-mass system. Smaller k lets the fluid compress more, which makes the fluid look unnaturally spongy.

In principle, k should be large, but if the time step is too large, then the oscillations grow and the system becomes unstable (and particles scatter abruptly). So, make k as large as possible while preserving stability at the desired time step. More on stability later.

Another equation of state, the [Tait equation](#), models liquids, which have lower compressibility:

$$p_i = \frac{k\rho_0}{\gamma} \left[\left(\frac{\rho_i}{\rho_0} \right)^\gamma - 1 \right]$$

Using this form can require small time steps and therefore more computation.

Note: You could make stiffness depend on an optimal local speed of sound and enforce stability for a given time step.

Pressure Gradient

Forces come from differences in pressure between two points—that is, *pressure gradients*. Combine the equation of state with an SPH gradient formula to compute the pressure gradient. For example:

$$\nabla p(\vec{r}_j) = \rho_j \sum_i \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla w(\vec{r}_{ij})$$

Evaluating ∇p at each particle provides the pressure term in the momentum equation, which tells you how to update particle velocity.

Clustering

Take another look at the smoothing kernels. As q diminishes from 1 to 0, the derivatives of the Gaussian, Lucy, and Poly6 kernels trough, then approach zero. Physically, this means that as particles get near each other, the pressure gradient initially pushes them apart; but when they become close enough, they feel less force pushing them apart and can become attracted. When they lie exactly on top of each other, they feel no separating force from each other, which allows particles to form spurious clusters. Using a different kernel to compute the pressure gradient can help avoid clustering. (See Schüssler & Schmitt [1981].)

The spiky kernel gradient does not go to zero at zero separation, but even when using it, particles near the boundary of a blob can form clusters.

You could avoid clusters by explicitly pushing apart particles that come too close. One way to do so involves computing another pressure gradient term. Clavet, Beaudoin, & Poulin (2005) achieved this by concocting a “near density” and “near pressure” using kernels with sharper falloff than their typical counterparts. They use these formulae, where q_{ij} is the distance between particles i and j , divided by an influence radius, h :

$$\rho_j = \sum_i (1 - q_{ij})^2$$

$$\begin{aligned}\rho_j^{near} &= \sum_i (1 - q_{ij})^3 \\ p_j^{near} &= k^{near} \rho_j^{near} \\ \nabla p_j &= \sum_i p_i (1 - q_{ij}) \\ \nabla p_j^{near} &= \sum_i p_i^{near} (1 - q_{ij})^2\end{aligned}$$

This entails keeping track of the usual and near quantities separately.

Dehnen & Aly (2012) analyze clustering (they call it the *pairing instability*) and find kernels that avoid it.

Viscous Diffusion and Dissipation

Viscosity (a fluid's resistance to deformation caused by stress) characterizes whether a fluid seems thick or thin. Although thick fluids have their use, thin fluids have more interesting flow. It would be nice if your fluid simulation could handle any range of viscosity.

Viscosity has another use: numerical stability. Remember that using a large stiffness constant avoids unnatural compression, but if k is too large, oscillations in the simulation grow until particles blast off. Viscosity damps oscillations and helps stabilize the simulation.

High viscosity makes fluids thick and syrupy, but if you wanted to simulate a thin fluid (like water), it would be nice if you could stabilize the simulation without cranking up viscosity too far. You can accomplish something like that by segregating viscous forces into two components: shear and shock.

You can model viscous diffusion as an exchange of velocity between particles. [Part 3](#) explained how to use Particle Strength Exchange to diffuse vorticity, and [part 10](#) explained how to use the same technique for diffusing heat. The same idea applies here, except for velocity:

$$\frac{d\vec{v}_j}{dt}_{shear\ diffusion} = \nu \sum_{i=1} (\vec{v}_i - \vec{v}_j)$$

But it turns out that the instability resulting from stiffness comes from oscillations directly along the line separating each particle in the pair. This makes sense because the pressure gradient operates along that line. You can damp the component of velocity directly between two approaching particles without damping other components. This lets each particle retain more velocity; hence, the motion does not dampen out as quickly.

You can tell whether two particles are approaching by computing the dot product between their relative velocity $\vec{v}_{ij} = \vec{v}_i - \vec{v}_j$ and their separation, \vec{r}_{ij} . If $\vec{v}_{ij} \cdot \vec{r}_{ij} < 0$; then, the particles are approaching, in which case diminish the component of their relative velocity parallel to \vec{r}_{ij} (see Figure 4). Artificial viscosity can take this form, where v_{av} is a viscosity parameter and ε is a small parameter to avoid singularity (divide-by-zero):

$$\frac{d\vec{v}_j}{dt}_{\text{artificial viscosity}} = -v_{av} \sum_{i=1} \frac{\vec{v}_{ij} \cdot \vec{r}_{ij}}{r_{ij}^2 + \varepsilon h^2} \nabla w(\vec{r}_{ij})$$

Note: This special form resembles von Neumann-Richtmyer artificial viscosity, which was invented in 1950 to reduce shock oscillations in fluid simulations. Shocks form when fluid moves faster than the speed of sound.

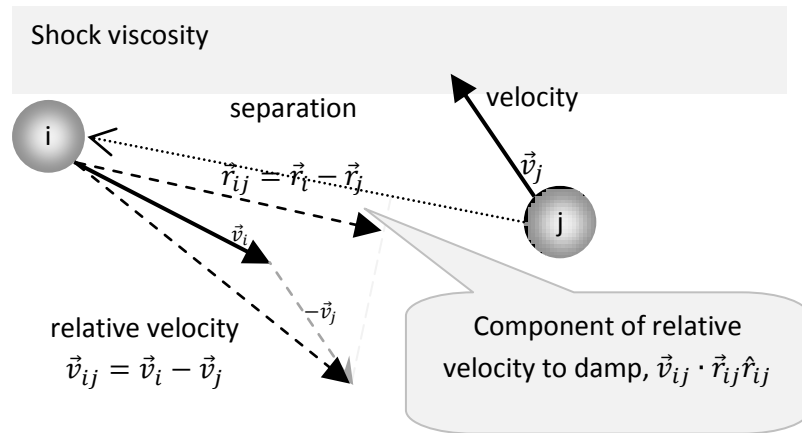


Figure 4. Shock viscosity

This implementation takes another approach: For each particle pair, compute a target radial velocity, which is the average of their previous radial velocity. It applies a difference to the velocities of each particle in a pair to make their radial velocities approach that target.

```
Vec3 &    velA    = mParticles[ idxA ].mVelocity ;
const Vec3 sep    = mParticles[ idxA ].mPosition - mParticles[ idxB ].mPosition ;
const float dist2 = sep.Mag2() ;
if( dist2 < mInflRad2 )
{
    // Particles are near enough to exchange velocity.
    const float dist = fsqrtf( dist2 ) ;
    const Vec3 sepDir = sep / dist ;
    Vec3 &    velB    = mParticles[ idxB ].mVelocity ;
    const Vec3 velDiff = velA - velB ;
    const float velSep = velDiff * sepDir ;
    if( velSep < 0.0f )
    {
        // Particles are approaching.
        const float infl = 1.0f - dist / mInfluenceRadius ;
        const float velSepA = velA * sepDir ;
        const float velSepB = velB * sepDir ;
        const float velSepTarget = ( velSepA + velSepB ) * 0.5f ;
        const float diffSepA = velSepTarget - velSepA ;
        const float changeSepA = mRadialViscosityGain * diffSepA * infl ;
        const Vec3 changeA = changeSepA * sepDir ;
        velA += changeA ;
        velB -= changeA ;
    }
}
```

// vel of pcl A along sep dir.
// vel of pcl B along sep dir.
// target vel along sep dir.
// Diff btw A's vel and target.
// Amount of vel change to apply.
// Velocity change to apply.
// Apply velocity change to A.
// Apply commensurate change to B.

Some kinetic energy between colliding particles dissipates to become heat. For simplicity, you can use a parameter, $\gamma_{dissipation}$, to govern dissipation:

$$\frac{d\vec{v}_j}{dt}_{dissipation} = -\gamma_{dissipation}\vec{v}_j$$

To conserve energy, particles could increase their heat to match this loss in kinetic energy. For visual effects, the amount of heating is usually negligible. But feel free to experiment.

The simulation applies constraints such as $v dt < 1$ and $\gamma_{dissipation} dt < 1$ so that diffusion and dissipation do not reverse the velocity direction for any particle.

Body Force

Fluid particles also experience a body force because of gravity. In lieu of explicitly needing to populate the entire simulation domain with fluid particles, model buoyancy as proportional to the difference of the particle mass density, $\rho_{m,j}$, and the density of fluid surrounding it, $\langle\rho_{m,i}\rangle$. In the absence of fluid particles (or for simplicity), use the ambient density for $\langle\rho_{m,i}\rangle$.

$$\frac{d\vec{v}_j}{dt} = g \frac{\rho_{m,j} - \langle\rho_{m,i}\rangle}{\rho_{m,j}}$$

Interaction with Bodies

To model fluid-body interactions, you can bounce particles off rigid bodies and exchange impulses. The same collision *detection* code used for vortons (see [part 4](#) and [part 13](#)) applies here. The collision *response* for SPH particles is similar to that applied to tracer particles:

```
static const float sElasticity      = 0.01f ;
static const float sImpactCoefficient = 1.0f + sElasticity ;
For each body...
const Vec3 & physObjVelocity      = rigidBody->GetVelocity() ;
For each particle...
// Contact point relative to body comes from collision detection.
const Vec3 vVelDueToRotAtConPt = rigidBody->GetAngularVelocity() ^ vContactPtRelBody ;
const Vec3 vVelBodyAtConPt     = physObjVelocity + vVelDueToRotAtConPt ;
const Vec3 velRelative         = particle.mVelocity - vVelBodyAtConPt ;
const float speedNormal        = velRelative * contactNormal ; // Contact normal depends on geometry.
const Vec3 impulse            = - speedNormal * contactNormal ; // Minus: speedNormal is negative.
particle.mVelocity = particle.mVelocity + impulse * sImpactCoefficient ;
```

Because a particle can lie behind multiple faces of a polytope container, repeat collision detection and response multiple times until either the particle no longer lies behind a face or up to some prescribed limit, such as 3.

Implementation Details

The simulation needs to compute the SPH summations for density, pressure, pressure gradient, and diffusion. A straightforward implementation would visit every pair, which would take $O(N^2)$ time, which is too slow:

```
for( int i = 0 ; i < N ; ++ i ) // For each particle...
  for( int j = i+1 ; j < N ; ++ j ) // For each OTHER particle...
    // operate on particle pair [i] , [j]
```

Because the smoothing kernel has compact support (limited range), you only need to consider nearby neighbors to each particle. A spatial partition can accelerate that search.

You can reuse the uniform grid from earlier articles in this series, but feel free to experiment with other formulations, such as octrees, spatial hashing (e.g., Worley [1996] and Teschner et al. [2003]), or Z-indexing (e.g., Goswami et al. [2010]).

Spatial partitioning can also help when parallelizing the algorithm. The spatial partition can lead to a natural partition of the data across multiple threads, such that each thread does not contend for access to the same data.

Visitation Algorithm

When computing the SPH sums, each particle's neighbors are visited. When using a uniform grid (or spatial hashing that indexes space according to a notional uniform grid), that code would look like this:

For each grid cell...

- For each neighboring cell...
 - Visit each particle.

Unfortunately, this would visit each particle pair twice. Ideally, each pair of particles should be visited only once.

Although it unnecessarily visits too many pairs, the naïve $O(N^2)$ algorithm only visits each pair once. Notice that the inner loop only visits particles with indices following those of the outer loop index. That's a clue! You could visit only those neighboring cells that "follow" the current cell.

The inner loop visits cells labeled **F** and **S** in Figure 5. Cells labeled **B** and **2** are skipped because when the outer loop visits those cells, they accessed the cell labeled **0** (which is the currently selected by the outer loop).

Cell **0** is handled separately; it uses the direct summation algorithm that visits each unique particle pair within that cell.

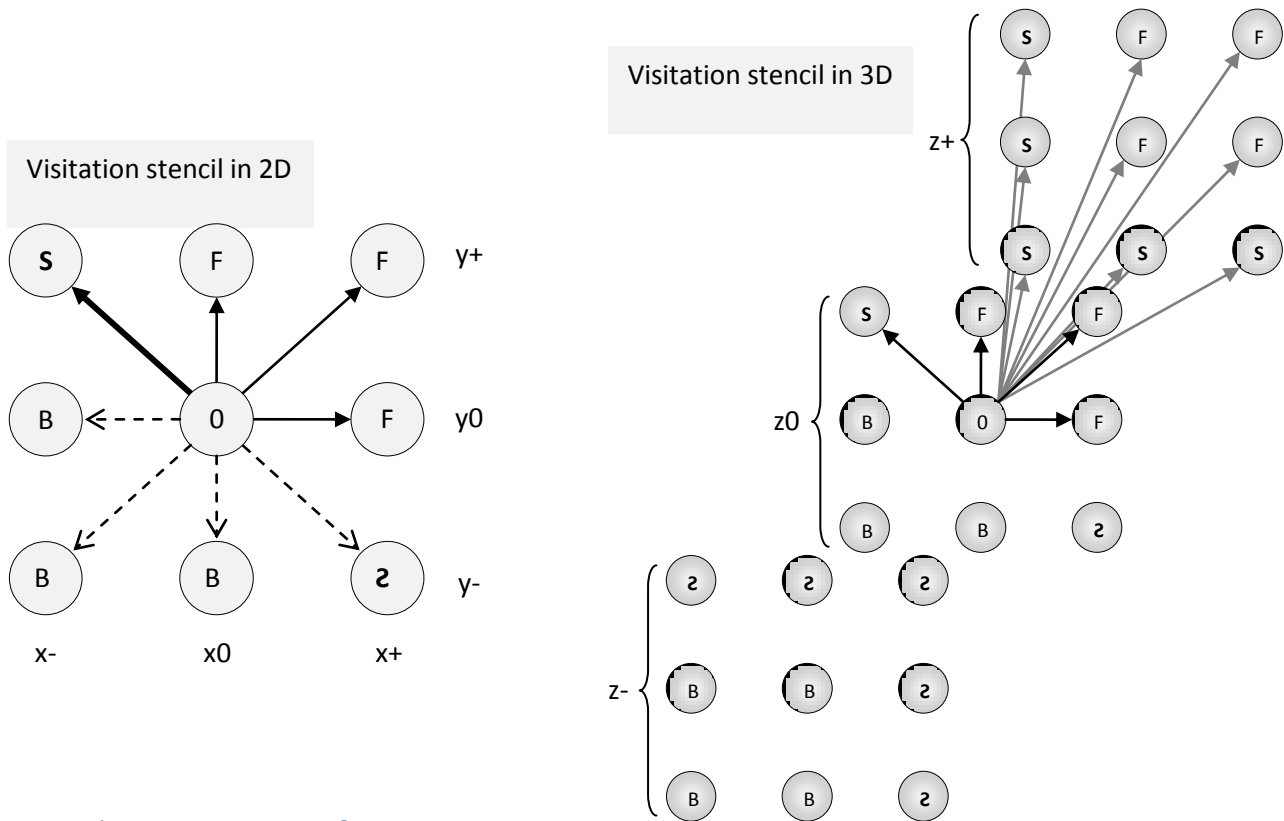


Figure 5. Visitation stencils

The routines that iterate through neighboring cells use a table to compute offsets relative to each outer-loop cell.

```
const int neighborCellOffsets[] =
{ // Offsets to neighboring cells whose indices exceed this one:
  1 // +, 0, 0 ( 1)
, -1 + nx // -, +, 0 ( 2)
, + nx // 0, +, 0 ( 3)
, 1 + nx // +, +, 0 ( 4)
, -1 - nx + nxy // -, -, + ( 5)
, - nx + nxy // 0, -, + ( 6)
, 1 - nx + nxy // +, -, + ( 7)
, -1 + nxy // -, 0, + ( 8)
, + nxy // 0, 0, + ( 9)
, 1 + nxy // +, 0, + (10)
, -1 + nx + nxy // -, 0, + (11)
, nx + nxy // 0, +, + (12)
, 1 + nx + nxy // +, +, + (13)
};
static const size_t numNeighborCells = sizeof( neighborCellOffsets ) / sizeof( neighborCellOffsets[ 0 ] );
```

This approach has one complication: It assumes that cells with all the neighboring indices exist and are actually neighbors. But that assumption is not true on boundaries (such as $i=0$ and $i=n-1$ for each axis). There are two outlier cases:

- ❑ **The offset is in bounds but not an actual neighbor.** This is harmless because none of the particles in those cells lie within the smoothing kernel cutoff, h . It wastes some computation but not enough to justify complicating the code to avoid those cells.
- ❑ **The offset is out of bounds.** This would be catastrophic because it would access memory illegally. Fortunately, that's easy to detect and handle. If the offset is outside the valid range of offsets, the code skips that offset. Furthermore, if any offset is invalid, all subsequent neighbor offsets will also be invalid (because the table orders them that way), so there is no need to attempt any subsequent offsets.

This is a case of easier done than said. The code to handle this is simple:

```
for( idx[2] = izStart + izShift ; idx[2] < izEnd ; idx[2] += zInc )
for( idx[1] = 0 ; idx[1] < nym1 ; ++ idx[1] )
for( idx[0] = 0 ; idx[0] < nxm1 ; ++ idx[0] )
{
    // For each grid cell...
    const size_t offsetX0Y0Z0 = idx[0] + idx[1] * nx + idx[2] * nxy;
    const size_t numInCurrentCell = pcIndicesGrid[ offsetX0Y0Z0 ].Size();
    for( unsigned iPcl = 0 ; iPcl < numInCurrentCell ; ++ iPcl )
    {
        // For each particle in this grid cell...
        const unsigned & pclIdx = pcIndicesGrid[ offsetX0Y0Z0 ][ iPcl ] ;
        for( size_t idxNeighborCell = 0 ; idxNeighborCell < numNeighborCells ; ++ idxNeighborCell )
        {
            // For each cell in neighborhood...
            const size_t celloffset = offsetX0Y0Z0 + neighborCellOffsets[ idxNeighborCell ] ;
            if( celloffset >= gridCapacity ) break ; // would-be neighbor is out-of-bounds.
            const VECTOR< unsigned > & cell = pcIndicesGrid[ celloffset ] ;
            for( unsigned iOther = 0 ; iOther < cell.Size() ; ++ iOther )
            {
                // For each particle in the visited cell...
                const unsigned & pclIdxOther = cell[ iOther ] ;
                operateOnParticlePair( pclIdx , pclIdxOther ) ;
            }
        }
    }
}
```

Note that the grid cells must have at least size h on each side for this $3 \times 3 \times 3$ neighborhood to include every possible neighbor.

To avoid having to pass arguments for all the data the particle pair operation needs, the invocation inside the loop body (represented by `operateOnParticlePair`) is a functor, an instance of one of these:

- ❑ `NumberDensityAccumulator` computes number density.
- ❑ `PressureGradientAccumulator` computes acceleration caused by the pressure gradient.
- ❑ `VelocityDiffuser` handles viscous diffusion and dissipation.

Parallelization

The loop over z (`idx[2]`) uses variables `izStart`, `izShift`, `izEnd`, and `zInc` to facilitate parallelization using Intel® Threading Building Blocks (Intel® TBB).

Because each cell updates particles in neighboring cells, processing is split into two phases: even and odd. When the outer loop addresses cells with even z-index values, the inner loop access cells with odd z-indices (and vice versa). To avoid race conditions, only one kind of outer loop runs per phase.

This code tells Intel TBB to compute number density using multiple threads:

```
void ComputeSphNumberDensityAtParticles_Grid( VECTOR< Vec2 > & fluidDensitiesAtPcls
, const VECTOR< Vorton > & particles
, const UniformGrid< VECTOR< unsigned > > & pclIndicesGrid )
{
    fluidDensitiesAtPcls.resize( particles.Size() , Vec2( 1.0f , 1.0f ) ) ;

    const unsigned & nz      = pclIndicesGrid.GetNumPoints( 2 ) ;
    const unsigned   nzm1    = nz - 1 ;

    // Estimate grain size based on size of problem and number of processors.
    const size_t grainSize = Max2( size_t( 1 ) , nzm1 / gNumberOfProcessors ) ;
    // Compute particle number density using threading building blocks.
    parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize ) ,
        SphSim_ComputeSphNumberDensityAtParticles_TBB( fluidDensitiesAtPcls , particles
            , pclIndicesGrid , VortonSim::PHASE_EVEN ) ) ;
    parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize ) ,
        SphSim_ComputeSphNumberDensityAtParticles_TBB( fluidDensitiesAtPcls , particles
            , pclIndicesGrid , VortonSim::PHASE_ODD ) ) ;
}
```

Results

The code accompanying this article has test cases that demonstrate SPH.

Falling

Figure 1 showed a column of fluid falling, sloshing to the left, then to the right. As the wave peak travels right, it crests, then breaks.

Sloshing

Figure 6 shows fluid sloshing inside a rotating container.

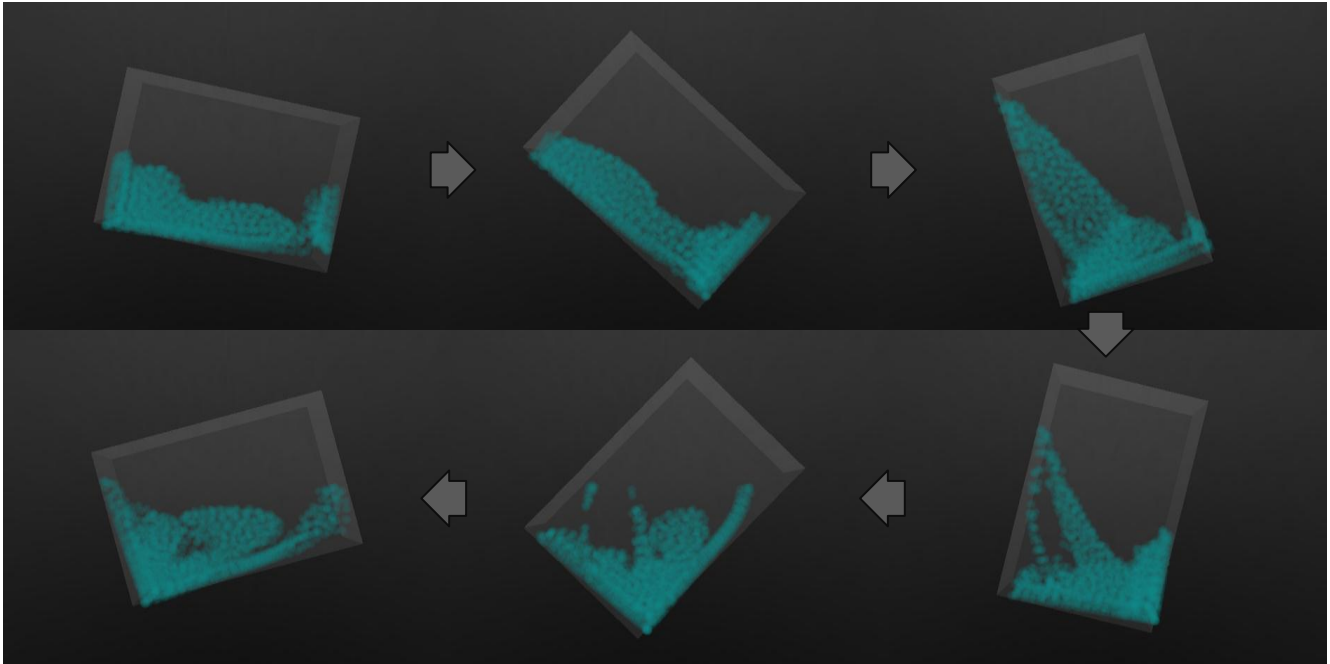


Figure 6. Fluid sloshing in a rotating container

Performance

Table 3 shows runtime performance for the sloshing SPH simulation, which had 1016 particles running on an Intel® Core™ i7 2600, which has four cores. Notice that the gains from multithreading seem modest. That is probably because the number of SPH particles is so small and the operations so simple that most of the runtime is consumed with either memory bandwidth or system overhead, such as thread creation and joining.

Table 3. CPU runtimesⁱ for SPH operations

# of threads	Density	Pressure	Diffusion	Total sim
1	1.60	3.18	1.44	6.41
2	1.49	2.96	1.36	6.02
3	1.28	2.45	1.14	5.09
4 (# cores)	1.18	2.30	1.07	4.76
6 (hyperthread)	1.17	2.29	1.06	4.73
8 (hyperthread)	1.14	2.25	1.03	4.61

To mitigate memory bandwidth bottlenecks, more operations could be consolidated. Density must be assigned for all particles before pressure can be computed, so those must run in separate phases. But pressure gradient, diffusion, and body forces could all run in a single routine. This would make the code less modular, but it might be worth the performance gains.

The number of particles is small. If the simulation used more particles, threading would have less overhead and probably be more efficient.

Summary

Smoothed particle hydrodynamics is a fluid particle simulation approach that directly solves for velocity. Compared to vortex methods, SPH more easily handles interaction with containers.

Although the ideas and code behind SPH are simple, applying them has numerous practical issues. The classic models have problems with particle clustering, so you must take special care to avoid that. Particles are kept apart by a repulsive force. To maintain numerical stability with large time steps, that force cannot be as stiff as you would want, so the fluid compresses and motion is springy. SPH also requires artificial viscosity to maintain stability, so SPH is ill suited to thin fluids with wispy motion like smoke but works better for thick and goopy liquids like oil or blood. The various simulation parameters are sensitive to each other and to other parameters of the fluid simulation, such as gravity and particle mass.

Future Articles

Perhaps there is a way to harvest how well SPH interacts with containers, with a vortex particle method's ability to avoid diffusion. It would be nice to find a way to avoid the springy compression artifacts of SPH without using an expensive iterative solver.

The particle rendering used in the figures and accompanying code is poor for liquids. It would be better to render only the liquid surface. To do that, you would need a surface tracking and extraction algorithm. The same information could be used to model surface tension.

Particle methods, including SPH and vortex, rely on spatial partitioning to accelerate neighbor searches. The uniform grid used in this series is simplistic and not the most efficient, and populating it takes more time than it should. It would be worthwhile to investigate various spatial partition algorithms to see which runs fastest, especially with the benefit of multiple threads.

Future articles will investigate these questions.

Further Reading

There are probably hundreds (maybe thousands) of articles on SPH. This list represents a tiny fraction of the available material:

- ❑ Gingold, R. A., and Monaghan, J. J. (1977). Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181.
 - The first paper to describe and name SPH.
- ❑ Lucy, L. B. (1977). A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82(12).
 - Another paper describing this particle method, which was in preparation at the same time as Gingold & Monaghan.
- ❑ Schüssler, M., & Schmitt, D. (1981). Comments on smoothed particle hydrodynamics.
 - Describes the clustering problem and proposes kernels that avoid it.
- ❑ Monaghan, J. J. (1992). Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30.
 - An early, 32-page survey article by one of the authors of SPH. Also see his second survey article from 2005.
- ❑ Hoover, W. G., Pierce, T. G., Hoover, G., Shugart, J. O., Stein, C. M., & Edwards, A. L. (1994). Molecular dynamics, smoothed-particle applied mechanics, and irreversibility. *Computers & Mathematics with Applications*, 28.
 - Introduced the cusp kernel, later called *spiky*.
- ❑ Stam, J., & Fiume, E. (1995). Depicting fire and other gaseous phenomena using diffusion processes. *SIGGRAPH*.
 - Uses a technique like SPH, in that it uses Monte Carlo integration, but it uses irregularly shaped blobs instead of spherical particles.
- ❑ Worley, S. (1996). A cellular texture basis function. *SIGGRAPH*.
 - Presents a 3D spatial hash later applied by others to SPH.
- ❑ Desbrun, M., & Gascuel, M-P. (1996). Smoothed particles: a new paradigm for animating highly deformable bodies. *Computer Animation and Simulation*.
 - An application of SPH to computer animation.
- ❑ Müller, M., Charypar, D., & Gross, M. (2003). Particle-based fluid simulation for interactive applications. *Eurographics*.
 - Introduced the poly6 kernel. See also Dehnen & Aly (2012).
- ❑ Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., & Gross, M. (2003). Optimized spatial hashing for collision detection of deformable objects. *Visualization and Machine Vision, Munich*.
 - Presents a spatial hashing algorithm that others later applied to SPH particle neighbor queries.
- ❑ Li, S., & Liu, W. K. (2004). *Meshfree particle methods*. Berlin: Springer-Verlag.

- Contains a chapter on SPH.
- ❑ Clavet, S., Beaudoin, P., & Poulin, P. (2005). *Particle-based viscoelastic fluid simulation*. Eurographics Symposium on Computer Animation.
 - Concocted the near-density/near-pressure formulation to avoid clustering. See also Dehnen & Aly (2012).
- ❑ Monaghan, J. J. (2005). Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68.
 - Another comprehensive (57-page) survey article by one of the original authors of SPH. Includes more than 120 references, dozens of which the author wrote or co-wrote. He appears to have written yet another survey article, “Smoothed particle hydrodynamics and its diverse applications,” in the *Annual Review of Fluid Mechanics*, vol. 44, January 2012.
- ❑ Colin, F., Egli, R., & Lin, F. Y. (2006). Computing a null divergence velocity field using smoothed particle hydrodynamics. *Journal of Computational Physics*, September.
 - Contrasts the difference gradient formula with the canonical and symmetric formulae.
- ❑ Goswami, P., Schlegel, P., Solenthaler, B., & Pajarola, R. (2010). Interactive SPH simulation and rendering on the GPU. *Eurographics*.
 - Uses Z-Indexing for the spatial partition for neighbor search.
- ❑ Madams, T. (2010, December 14). Why my fluids don’t flow [Web log post]. Retrieved from <http://imdoingitwrong.wordpress.com/tag/sph>
 - Chronicles the perils of trying to implement algorithms described in academic literature. Includes source code and nice videos.
- ❑ Dehnen, W., & Aly, H. (2012). Improving convergence in smoothed particle hydrodynamics simulations without pairing instability. *Monthly Notices of the Royal Astronomical Society*, June.
 - Analyzes several smoothing kernels to find which avoid clustering.

About the Author

Dr. Michael J. Gourlay works as a senior software engineer on interactive entertainment. He previously worked at Electronic Arts (EA Sports) as the software architect for the Football Sports Business Unit, as a senior lead engineer on *Madden NFL*, on character physics and the procedural animation system used by EA on *Mixed Martial Arts*, and as a lead programmer on *NASCAR*. He wrote the visual effects system used in EA games worldwide and patented algorithms for interactive, high-bandwidth online applications. He also developed curricula for and taught at the University of Central Florida, Florida Interactive Entertainment Academy, an interdisciplinary graduate program that teaches programmers, producers, and artists how to make video games and training simulations. Prior to joining EA, he performed scientific research using computational fluid dynamics and the world’s largest massively parallel supercomputers. His previous research

also includes nonlinear dynamics in quantum mechanical systems and atomic, molecular, and optical physics. Michael received his degrees in physics and philosophy from Georgia Tech and the University of Colorado Boulder.

Intel, the Intel logo, and Core are trademarks of Intel Corporation in the US and/or other countries.

Copyright © 2013 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.