Fluid Simulation for Video Games (part 14)

By Dr. Michael J. Gourlay

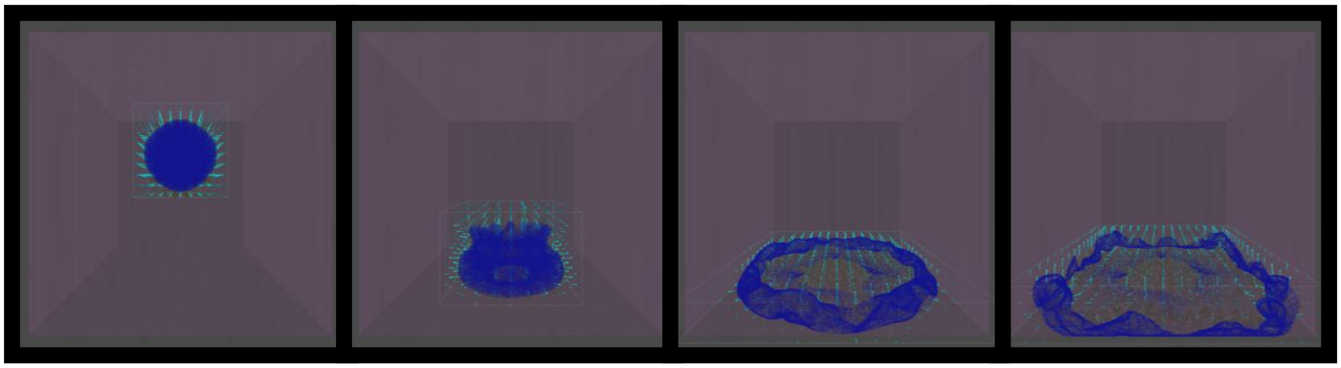


Figure 1. *Dyed fluid drop inside convex polyhedral container. Cyan arrows show the density gradient. Yellow balls show vortex particles (vortons).*

Containers

Liquids are one phase of fluids in that that they take the shape of their container on all sides but one. This article explains how to turn polyhedra inside out to make containers. It also demonstrates one way to use lock-free atomic operations, exposed through Intel® Threading Building Blocks (Intel® TBB), to allow thread-safe parallelization.

This article—the 14th in a series—adds rudimentary containers to the fluid particle system described earlier. 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.

Inside-out Polytopes

Detecting collisions between objects with arbitrary geometry requires sophisticated algorithms and mathematics, especially if you want to do that in real time. Fortunately, however, for visual effects, detecting collisions between particles and objects is much simpler, because you can treat particles as points or spheres. Furthermore, for video games visual effects (VFX), you can usually get away with colliding against the convex hull of an object rather than dealing with its detailed geometry.

Part 13 described how to collide particles against convex polyhedral solids. Let's turn those solids inside out to make convex polyhedral holes. These holes let you contain fluid particles, which is useful when modeling liquids.

Broad Phase: Bounding Sphere

As in part 13, this article uses planes to create a half-space model of a convex polyhedron.

Before jumping directly into computing the distance between points and the multiple planes that form a convex polytope, you should use a broad-phase bounding-volume test to avoid expensive computations. Broad-phase collision detection should be fast, and although it's okay for the broad-phase to detect particles that will not collide with the target, it must detect all that will. For a *solid,* a bounding sphere that circumscribes the target volume satisfies these requirements.

For testing whether a particle is inside a *hole,* however, instead of a sphere that circumscribes the outside, you need a sphere that inscribes the inside; any particles outside that sphere might collide with the exterior, and all particles inside the sphere definitely will not. Figure 2 shows an example of circumscribed and inscribed spheres. Inscribed spheres are to holes what circumscribed spheres are to solids.

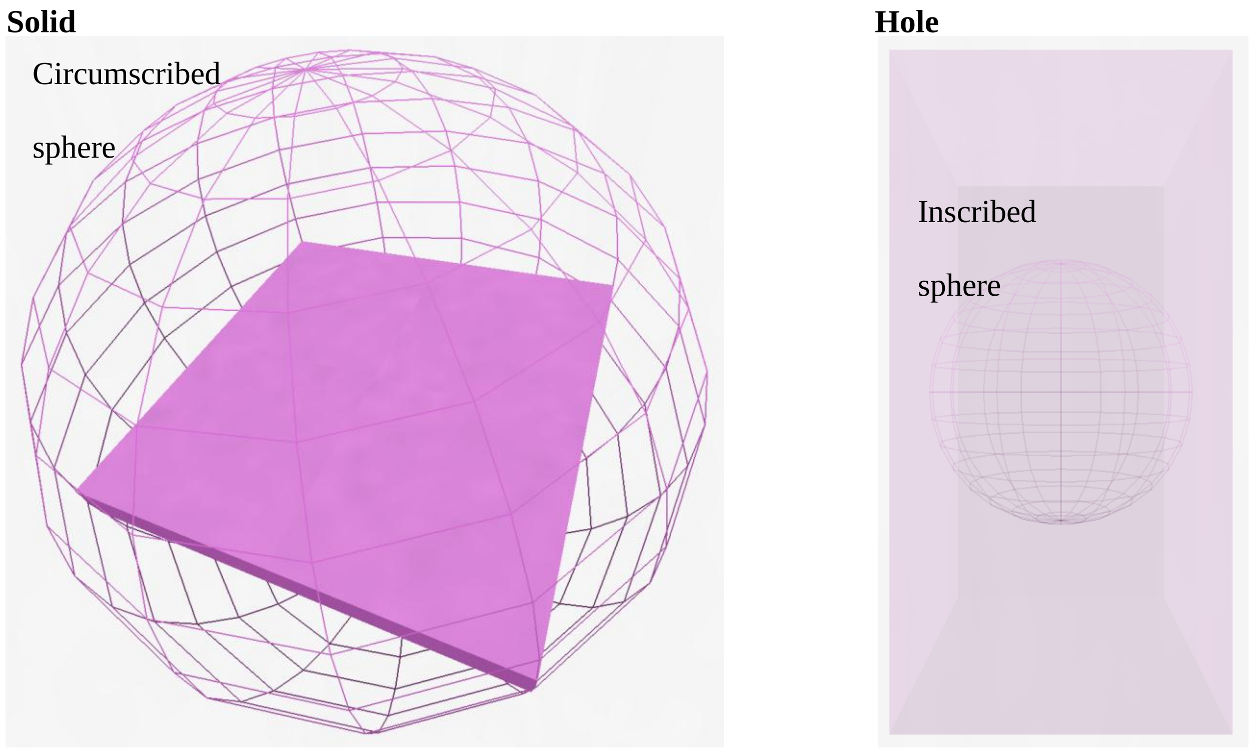


Figure 2. *Circumscribed and inscribed spheres serve as broad-phase collision volumes for solids and holes, respectively.*

Change `CollisionShape` to discriminate between solids and holes:

- Add a floating point member, `mParity`, which is `+1` for a solid and `-1` for a hole.
- Add a method, `IsHole`, that returns `false` for solids and `true` for holes.

The broad-phase tests in `FluidBodySim::CollideVortonsSlice` and `FluidBodySim::CollideTracersSlice` depend on that property. For example, `CollideTracersSlice` uses this test:

```
if( collisionShape->IsHole() )
{
    const float fCombinedRadii  = Max2( boundingRadius - rTracer.GetRadius() , 0.0f ) ;
    broadPhaseCollision = fSphereToTracer > fCombinedRadii ;
}
else
{
    const float fCombinedRadii  = ( boundingRadius + rTracer.GetRadius() ) ;
    broadPhaseCollision = fSphereToTracer < fCombinedRadii ;
}
```

Holey Polytopes: Inward-facing Planes

The narrow-phase collision detection tests also require some changes for holes. The change depends on whether you want to compute the distance between stationary or moving objects.

Contact Distance

First, consider stationary objects. As part 13 described, the distance of a point to a convex polytope is the same as the largest distance of the point to any feature of that polytope, where a feature could be a vertex, edge, or face. If you consider only faces, that will still tell you whether the point lies inside or outside the polyhedron, so to keep things simple, consider only faces.

Holes use exactly the opposite logic. Figure 3 compares collision detection for solids and holes. You can use identical logic to detect whether a point is inside or outside a hole simply by multiplying the point-to-plane distance by the parity.

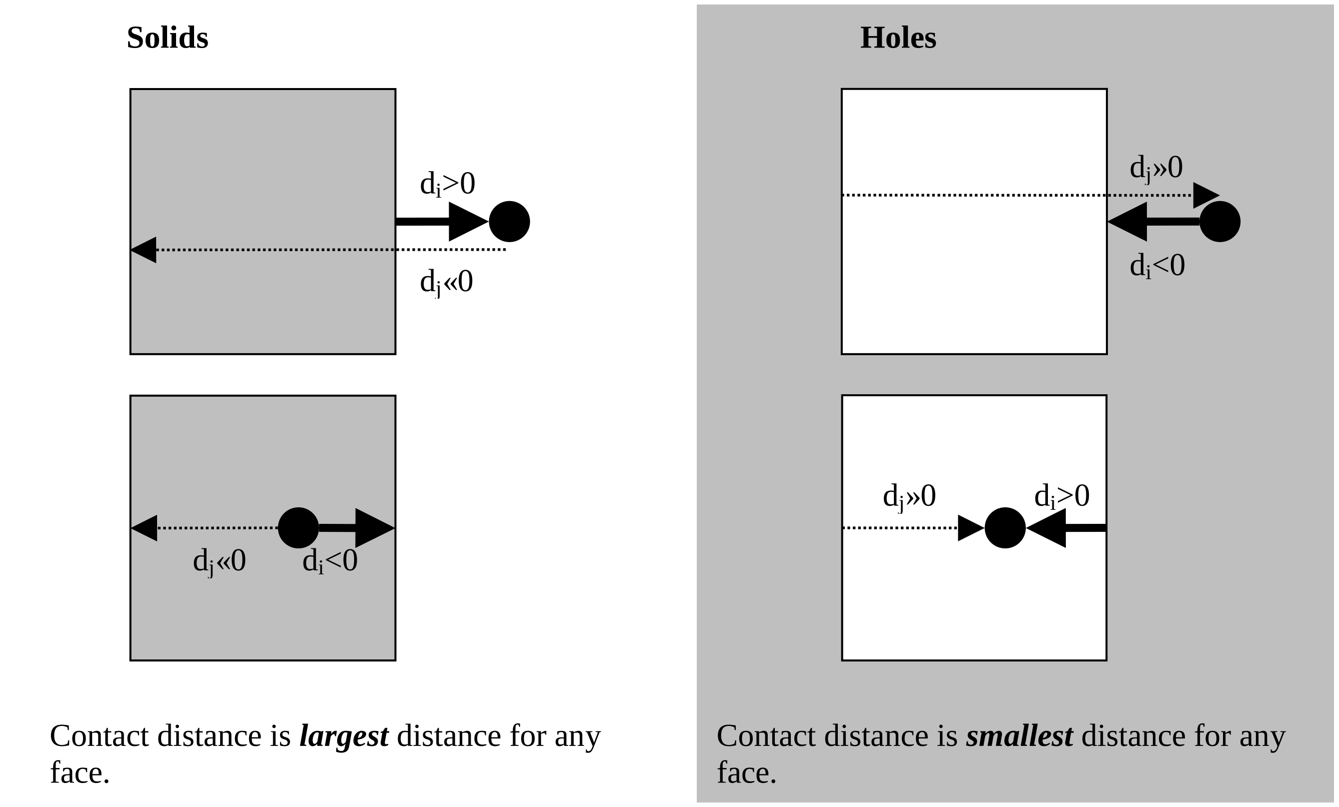

Solids
$d_i>0$
$d_j«0$
$d_j«0$
$d_i<0$
Contact distance is ***largest*** distance for any face.
Holes
$d_j»0$
$d_i<0$
$d_j»0$
$d_i>0$
Contact distance is ***smallest*** distance for any face.

Figure 3. *Computing the distance between a point and solid and hole polytopes*

This routine computes the point-to-plane distance for both solids and holes:

```
float ConvexPolytope::ContactDistance( const Vec3 & queryPoint
    , unsigned & idxPlaneLeastPenetration ) const
{
    float largestDistance = - FLT_MAX ;
    const size_t numPlanes = mPlanes.Size() ;
    for( unsigned iPlane = 0 ; iPlane < numPlanes ; ++ iPlane )
    {   // For each planar face of this convex hull...
        const float distToPlane = mPlanes[ iPlane ].Distance( queryPoint ) ;
        if( distToPlane > largestDistance )
        {   // Point distance to iPlane is largest of all planes visited so far.
            largestDistance = distToPlane ;
            // Remember this plane.
            idxPlaneLeastPenetration = iPlane ;
        }
    }
    return largestDistance * GetParity() ;
}
```

The code in purple bold above that is multiplying by parity shows the key aspect that lets this code work for both solids and holes.

Likewise, computing the contact point and normal direction also needs the parity factor:

```
Vec3 ConvexPolytope::ContactPoint( const Vec3 & queryPoint ,
  const unsigned idxPlaneLeastPenetration , const float distance ) const
{
  const Vec3 & contactNormal = mPlanes[ idxPlaneLeastPenetration ].GetNormal() * GetParity();
  Vec3         contactPoint  = queryPoint - contactNormal * distance ;
  return contactPoint ;
}
```

You could potentially make this code more elegant by changing the definition of a plane so that the `d` parameter is negative, thereby eliminating the need to use `GetParity` in these two routines. But I find it easier to make the solid-versus-hole more explicit, and the logic differs for moving objects anyway.

Collision Detection

For detecting collisions between moving objects, the difference between solids and holes is subtler and slightly more complicated. Figure 4 shows why. The difference arises when the point is behind a planar face. If a point lies inside a solid, it could hypothetically get pushed in any direction and escape the solid, so which direction is correct? For the solid case, the point could have traveled so far in a single update that even though it's closer to one face, it might be moving along that face normal—hence, not getting deeper. That implies that the point penetrated some other face. To determine which face the point is "most behind," compute $\vec{v} \cdot \hat{n}$, where $\vec{v}$ is the point velocity relative to the body and $\hat{n}$ is the surface normal direction. That way, when the point is ejected from the body, it goes in the correct direction—opposite its precollision direction.

For holes, the situation differs. If a particle is outside the hole, then the velocity is irrelevant; the most appropriate direction for the particle to move is toward the inside of the hole. That is true regardless of whether the particle was already moving that way. The situation is not perfectly analogous to the solid case, where the correct “ejection direction” depends on the particle velocity. For holes, the particle velocity does not determine in which direction to move the particle to resolve the collision.

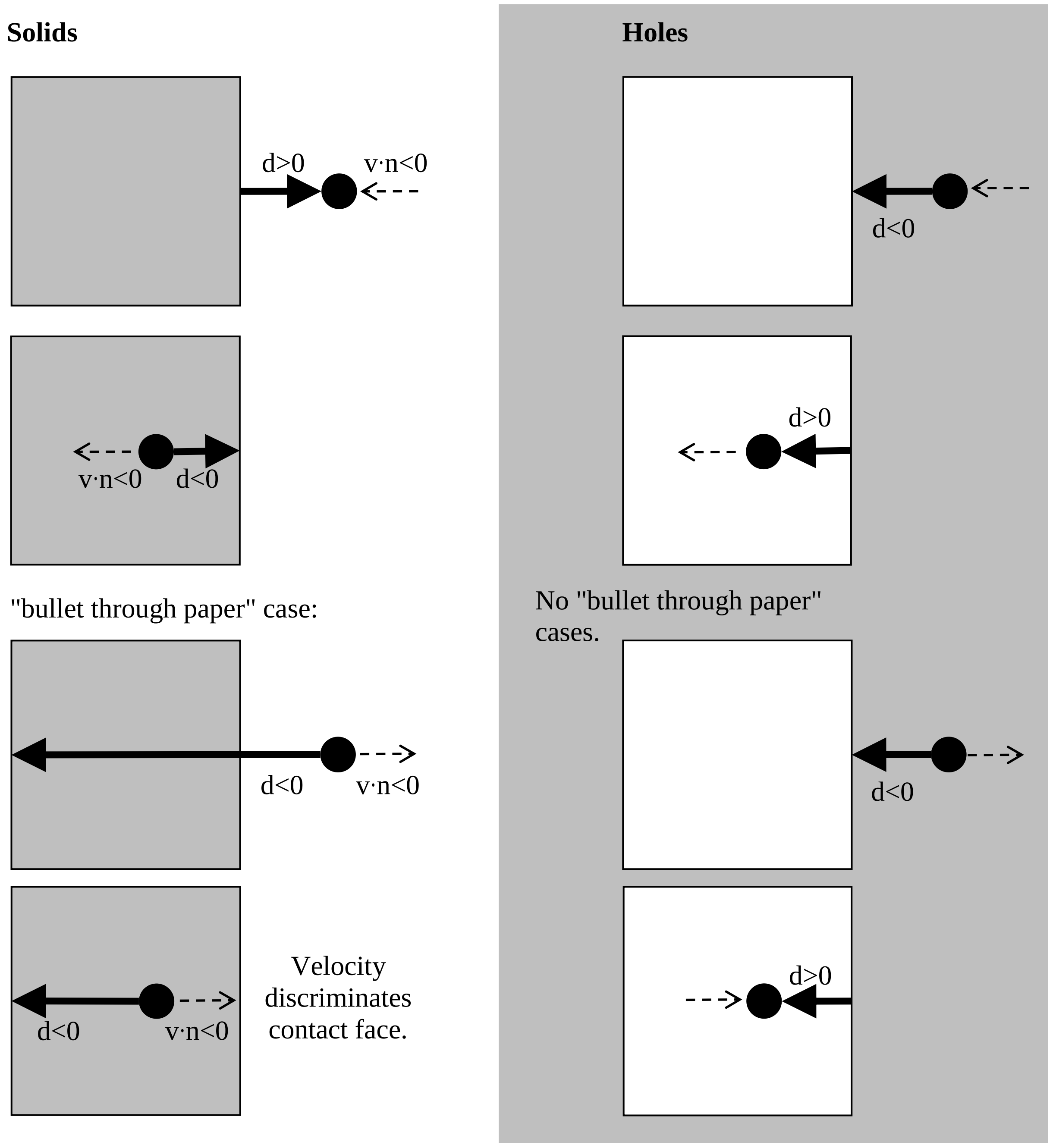

Solids
Holes
d>0
v·n<0
d<0
v·n<0
d<0
d>0
"bullet through paper" case:
No "bullet through paper" cases.
d<0
v·n<0
d<0
d<0
v·n<0
d>0
Velocity discriminates contact face.

Figure 4. *Detecting collisions between a point and exterior and interior polytopes*

This routine computes the collision distance between a point and convex polyhedron:

```
float ConvexPolytope::CollisionDistance( const Vec3 & queryPoint
  , const Vec3 & queryPointRelativeVelocity
  , unsigned & idxPlaneLeastPenetration ) const
{
    float largestDistance = - FLT_MAX ;
    const size_t numPlanes = mPlanes.Size() ;
    for( unsigned iPlane = 0 ; iPlane < numPlanes ; ++ iPlane )
    {   // For each planar face of this convex hull...
        const Math::Plane & testPlane = mPlanes[ iPlane ] ;
        const float distToPlane = testPlane.Distance( queryPoint ) ;
        if( distToPlane > largestDistance )
        {   // Point distance to iPlane is largest of all planes visited so far.
            const float speedThroughPlane =
                        queryPointRelativeVelocity * testPlane.GetNormal() * GetParity() ;
            if(     ( speedThroughPlane <= 0.0f ) // Query point is going deeper thru face.
                ||  ( distToPlane >= 0.0f )       // Query point is outside polytope.
                ||  ( IsHole() )                  // Polytope is a hole.
                )
            {   // Query point is moving deeper through this plane.
                largestDistance = distToPlane ;
                // Remember this plane.
                idxPlaneLeastPenetration = iPlane ;
            }
        }
    }
    return largestDistance * GetParity() ;
}
```

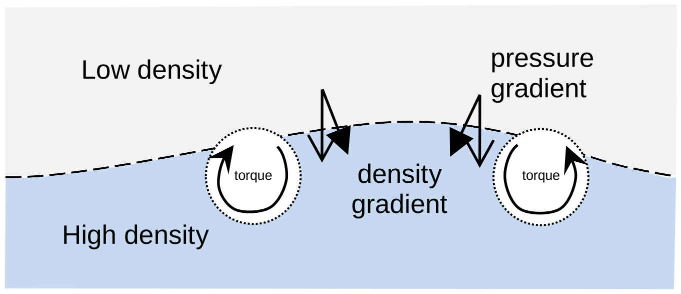
Note the use of `IsHole` and `GetParity`, in purple bold, to handle holes.

Pressure at Walls

If you made only the changes above to collision detection and put vortex particles in a hole, you would get nonsense. The trouble started back in parts 4 and 8.

Heavy fluid sinks, and light fluid rises. To capture that behavior, the fluid dynamics equations need to depend on density variation. But in the most general form, density variation includes sound waves, which are expensive to simulate and provide no improvement to VFX. The Boussinesq approximation dodges that problem by assuming that the fluid is in hydrostatic equilibrium, meaning that the only pressure gradient is the result of the weight of a stack of fluid pressing down on itself.

That idea worked well enough for unbounded fluids, as parts 8, 9, 10, and 11 demonstrated. But fluids remain in containers because the walls push back, as shown in Figure 5. Such a flow has multiple domains (separated by rigid body walls) across which density and pressure jump discontinuously. The “Further Reading” section in this article describes recent research into this interesting problem.



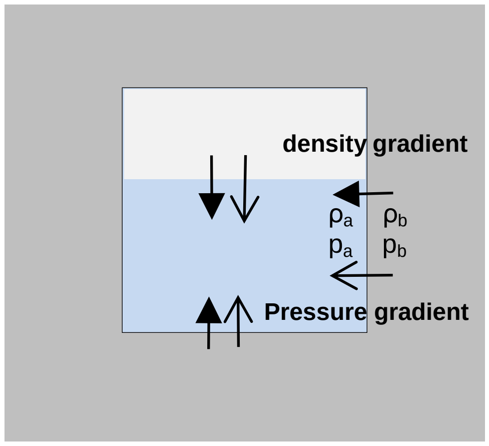


Figure 5. *Density and pressure gradients at boundaries.*

For the sake of brevity, however, this article postpones any reformulation of the simulation formulae and instead takes an easy way out. Let's see what we can hack up.

The immediate problem is that the baroclinic term is not zero when it should be zero:

$$\underbrace{\frac{D\vec{\omega}}{Dt}}_{\text{Change in Vorticity}} = \underbrace{(\vec{\omega}\cdot\vec{\nabla})\vec{v}}_{\text{Stretching \& tilting}} + \underbrace{\nu\nabla^2\vec{\omega}}_{\text{Viscous diffusion}} + \underbrace{\frac{\vec{\nabla}\rho\times\vec{\nabla}p}{\rho^2}}_{\text{Buoyancy}} + \underbrace{\vec{\tau}}_{\text{External torque}}$$

The simulation code uses the simplification $\vec{\nabla}p = \rho\vec{g}$ (where $\vec{g}$ points down) and so never computes $\vec{\nabla}p$. The baroclinic term gets computed like this:

```
    Vec3        densityGradient ;
    mDensityGradientGrid.Interpolate( densityGradient , rVorton.mPosition ) ;
    const Vec3  baroclinicGeneration = densityGradient ^ mGravAccel * oneOverDensity ;
```

At vertical walls, however, both pressure and density should jump horizontally. They should not generate vorticity when at equilibrium. But the simulation computes density on the grid, which does not incorporate the rigid body surface, nor does the mesh have sufficient resolution to do so. Furthermore, the simulation treats the pressure as being in hydrostatic balance and its gradient as being purely vertical. The spurious horizontal density gradient crosses with the vertical pressure gradient to create a spurious torque on the fluid.

Note: Fluid being in hydrostatic balance means that at a given height, the pressure is $p = p_0 + \rho g h$ inside the container. Pressure is a scalar—it has no direction—so it is *isotropic*—that is, the same in all directions. But its value abruptly changes across the wall, so its gradient there has a horizontal component. Representing that requires special treatment currently ignored in the simulation presented so far.

To model this correctly, fluid simulations adapt their mesh to conform to the boundaries and treat the jump explicitly. (For example, read up on the Reimann problem and Reimann solvers.) Future articles will revisit this problem.

In lieu of adding a separate pressure variable and precisely resolving boundaries, you could “poison” the density gradient at container boundaries so to prevent spurious baroclinic generation at walls. You could introduce this poison in at least two ways:

- Record which vortons hit walls. Then, reset the density gradient of any grid cell containing such vortons to have zero component parallel to surface normals. The simulation already detects when vortons hit walls so. But collision detection runs after computing the baroclinic term (and those operations cannot easily be reordered), so there is a small latency.
- Directly determine which grid points lie inside walls and reset their density gradient to have zero along surface normals. This has no latency but requires passing information between the `VortonSim` and the `FluidBodySim`.

Both approaches have benefits and drawbacks. Try both.

This first approach uses “hit” information to poison grid cells near boundaries:

```
static void PoisonDensityGradientSlice( UniformGrid< Vec3 > & densityGradientGrid
    , const Vector< Vorton > & particles , size_t iPclStart , size_t iPclEnd )
{   // Poison density gradient grid based on vortons in contact with container boundaries.
    const Vec3 zero( 0.0f , 0.0f , 0.0f ) ;
    for( size_t iParticle = iPclStart ; iParticle < iPclEnd ; ++ iParticle )
    {   // For each particle in the array...
        const Particle  &   rParticle   = particles[ iParticle ] ;
        if( rParticle.mHitBoundary )
        {   // This particle hit a boundary.
            const Vec3      &   rPosition   = rParticle.mPosition   ;
            // Zero out any gradients along contact normal.
            // Remove normal component at each surrounding gridpoint.
        #if USE_TBB
          densityGradientGrid.RemoveComponent_ThreadSafe( rPosition , rParticle.mHitNormal );
        #else
          densityGradientGrid.RemoveComponent( rPosition , rParticle.mHitNormal ) ;
        #endif
        }
    }
}
```

Another routine, `FluidBodySim::PoisonDensityGradient`, determines which grid points reside inside walls and zeroes the horizontal density gradient there. This snippet shows the most interesting part (and you can see the whole routine in the accompanying code archive):

```
const float contactDistance = convexPolytope->ContactDistance( gridPointPosition
                                    , physObjPosition , physObjOrientation , idxPlane ) ;
if( contactDistance < testRadius )
{   // Gridpoint is in contact with rigid body.
    // Zero out any gradients along contact normal.
    const Vec3  vContactPtWorld = convexPolytope->ContactPoint( gridPointPosition
                    , physObjOrientation , idxPlane , contactDistance , contactNormal ) ;
    const float densGradAlongNormalMag  = densGrad * contactNormal ;
    const Vec3  densGradAlongNormal     = densGradAlongNormalMag * contactNormal ;
    densGrad -= densGradAlongNormal ;
}
```

As physics simulations go, this is a pretty bizarre hack. It causes important problems that need to be addressed, but as Alton Brown says, that's another show.

Parallelization

The `PoisonDensityGradient` routine above needs special attention for it to become thread safe. It writes to grid points depending on particle positions. The loop iterates over particles. It would therefore be most natural to partition across threads by particle index. But particle index has no relationship to particle position, so multiple threads could simultaneously write to the same grid point. That would be a race condition.

You could first spatially partition vortons into the grid, then parallelize the problem across a spatial coordinate, thereby avoiding contention. Indeed, the simulation already partitions particles for computing viscous and thermal diffusion. So that approach would work.

Intel® TBB provides another option: lock-free atomic operations. One of those operations is called `fetch_and_add` and does what you want: It atomically adds a value to a variable. Unfortunately, it only operates on integer types, and you need to operate on floats. Instead, you can use `compare_and_swap`. It's much slower than `fetch_and_add` it requires additional code, but it's faster and cheaper than using a mutex, because it requires neither a context switch nor additional memory for the lock.

Note: Beware that `AtomicUpdate` in Intel® TBB Design Patterns has at least two errors: The inner `int o` masks the outer `int o`; the inner assignment to `o` should have no type declaration. Also, the documentation indicates that the formal arguments to `compare_and_swap` are `newValue` and `comparand`, whereas `AtomicUpdate` passes the opposite; they should be reversed.

This utility routine provides similar functionality to `fetch_and_add` but works for floats:

```
inline void Float_FetchAndAdd( float & sum , const float & increment )
{   /// Atomically increment sum.
    tbb::atomic<float> & atomicSum = reinterpret_cast< tbb::atomic<float> & >( sum );
    float sumOld , sumNew ;
    do
    {
        sumOld = atomicSum ;
        sumNew = sumOld + increment ;
    } while( atomicSum.compare_and_swap( sumNew , sumOld ) != sumOld ) ;
}
```

The code archive accompanying this article provides a thread-safe implementations for `UniformGrid::RemoveComponent` that uses `Float_FetchAndAdd`, along with an Intel® TBB functor that wraps `PoisonDensityGradientSlice` for use with `tbb::parallel_for`.

Results

Consider a container about half-full of fluid particles, as shown in Figure 6.

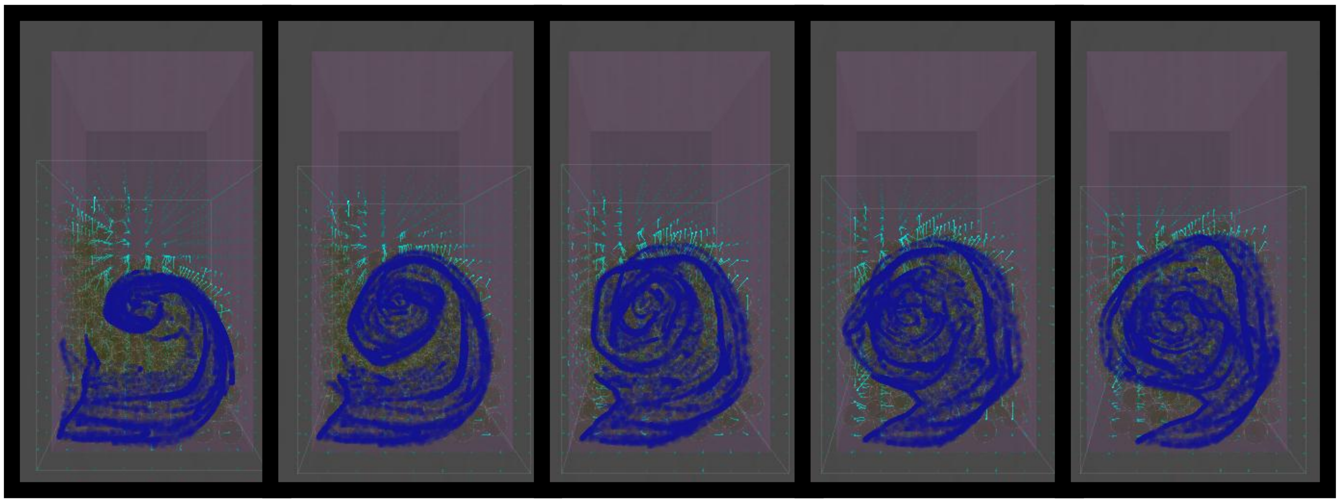


Figure 6. *Dyed fluid sloshing in a container moving left and right. Cyan arrows show the density gradient. Yellow balls show vortex particles (vortons).*

Performance

As expected, using Intel® TBB to parallelize `PoisonDensityGradient` speeds up that process by running it on multiple threads, but unfortunately, the naive implementation is not thread safe. It's a useful comparison, but its yields unreliable results. For the numbers of particles used in the demonstration scenario, the thread-safe `PoisonDensityGradient` routine that uses `compare_and_swap` runs more slowly than even the serial version. That is an artifact of the low number of particles and the extremely large overhead associated with using `compare_and_swap` along with its conditional branches. If the hardware supported `fetch_and_add` for floats, perhaps the runtime would diminish (see Figure 7).

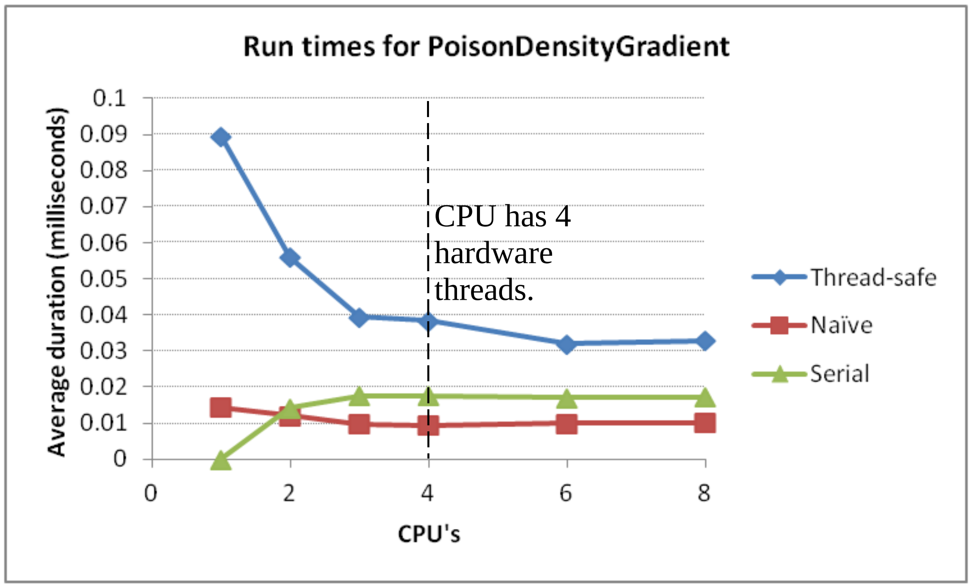


Figure 7. *Runtimes for* **`PoisonDensityGradient`**

Summary

This article showed how to turn convex polyhedra inside-out to make containers for particles using a half-space representation. For stationary particles, the technique for holes exactly mirrors for solids, but particles have an asymmetry, so holes require slightly different calculations than for solids.

Putting particles inside a container exposes issues that occur when the simulation has to resolve collisions from all sides. Ironically, getting a pile of particles to become and remain stationary is more difficult than getting them to swirl around each other. Although this article shows the geometry of how to keep particles contained, it opens a broader spectrum of problems, ripe for discussion in future articles.

Baroclinic generation is difficult to get right at walls. A fully accurate solution would overextend the scope of this article. So for brevity, this article described a quick-and-dirty way to eliminate spurious baroclinic torque by changing the density gradient near walls. That also exposed an opportunity to use `compare_and_swap`, a lock-free synchronization method.

Future Articles

Liquids have a free surface that gases lack. Simulating that surface requires a numerical model for surface tension. That, in turn, requires identifying and tracking the surface. Modeling surface tension also entails a physical model of intermolecular forces. And to visualize the surface requires a different rendering technique. Future articles will delve into these interesting challenges.

Further Reading

- Gino van den Bergen. (2003). Collision detection in interactive 3D environments. *The Morgan Kaufmann series in interactive 3D technology.*
    - A great book that covers distance algorithms, spatial data structures, and other topics relevant to this article. van den Bergen presents at the GDC physics tutorial session each year, and his slides are available online.
- Christer Ericson. (2005). Real-time collision detection.
    - This piece has tons of details about sophisticated collision-detection algorithms and includes a discussion of half-spaces and sphere-to-plane distance. It also discusses numerical robustness, an underrated topic that deserves more attention. Ericson also presented tutorials at GDC, for which the slides are available on his website.
- David Randall. (2010). The anelastic and Boussinesq approximations. *Quick Studies in Atmospheric Science.* Available at http://kiwi.atmos.colostate.edu/group/dave/pdf/AneBous.pdf.
    - This article describes the anelastic and Boussinesq approximations of the fluid dynamics equations and lists several articles that provide improvements. The goal of those approximations is to eliminate sound waves (pressure waves that cause numerical difficulties) while retaining some compressibility. The simulations in this article series use the Boussinesq approximation, which leads to difficulties prominently visible in the current article.
- Sharif Elcott, Yiying Tond, Eva Kanso, Peter Schroeder, and Mathieu Desbrun. (2007). Stable, circulation-preserving, simplicial fluids. *ACM Transactions on Graphics,* 26(1).
    - This article describes a clever vortex-based simulation in which the authors discretize space in simplices (triangles in 2D, tetrahedra in 3D). Their simulation is constructed to conserve circulation and hence lacks numerical diffusion of vorticity. (The simulation in this series also conserves linear and angular momentum because of explicit exchanges of those quantities between the fluid and objects.) Aside from being more rigorous, their simulation is more flexible in that it correctly simulates buoyant flow inside polygonal containers with arbitrary shape—even concave shapes. Their approach, however, relies on a grid and does not use particles. It would not especially suit video games without drastically changing the VFX authoring workflow.
- Sebastien Blaise, Richard Comblen, Vincent Legat, Jean-Francois Remacle, Eric Deleersnidjer, and Jonathan Lambrechts. (2010). A discontinuous finite element baroclinic marine model on unstructure prismatic meshes.
    - This article describes a fluid simulation that includes discontinuous jumps in density and pressure. Tuomas Karna's 2012 thesis, *Development of a baroclinic discontinuous Galerkin finite element model for estuarine and coastal flows,* covers a similar topic.

- ❑ Cesar Dopazo, Antonio Lozano, and Felix Barreras. (2000). Vorticity constraints on a fluid/fluid interface. *Physics of Fluids,* 12(8).
  - This brief communication derives exact relationships between vorticity on both sides of a liquid–gas interface, such as the surface of a fluid.

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.