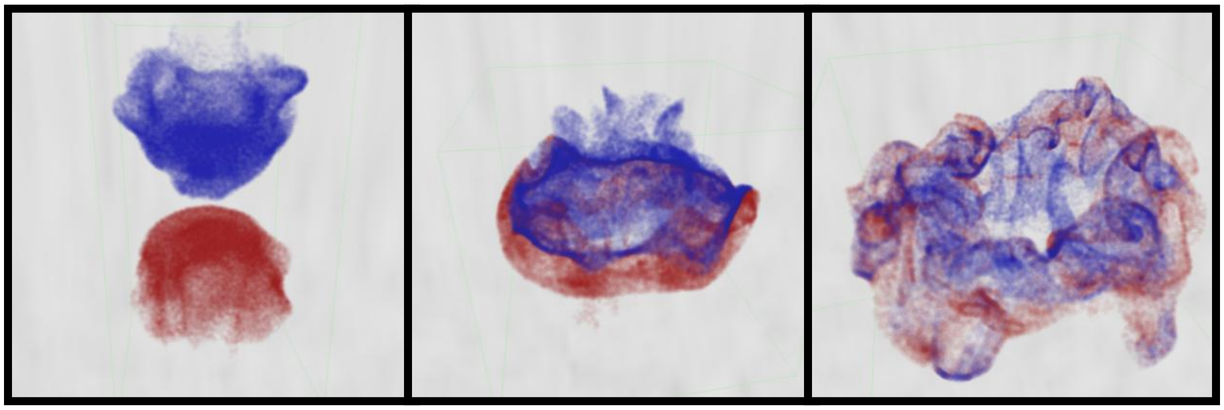
Fluid Simulation for Video Games (Part 8)

By Dr. Michael J. Gourlay



Baroclinicity: Fluid Buoyancy

Heavier fluid sinks, and lighter fluid rises.

This article, the eighth in a series, explains how a vortex-based fluid simulation handles variable density in a fluid. 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 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.

This article introduces features to the simulations presented in previous articles: The fluid flow includes motion because of buoyancy—heavier fluid sinks, and lighter fluid rises. These new features facilitate visual effects with effects of variable density or (with additional rendering work) multiple fluids, such liquid–gas mixtures like water and air. It also lays the groundwork for upcoming features like thermal convection (hot air rises) and combustion (generating heat from chemical processes).

Using Vortices to Push Heavy Fluid Below Light Fluid

Remember that the simulation presented in these articles focuses on vorticity. How does the vorticity equation express this effect of pushing heavy fluid underneath light fluid?

Remember from Part 1 that by taking the curl of the momentum equation, you derive the vorticity equation:

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

The buoyancy term expresses the tendency, depicted in Figure 1, to create regions of overturning, where fluids with density out of equilibrium (that is, heavy fluid above light fluid) form rolling currents that tend toward bringing the fluid into equilibrium (that is, vortices push heavy fluid below light fluid).

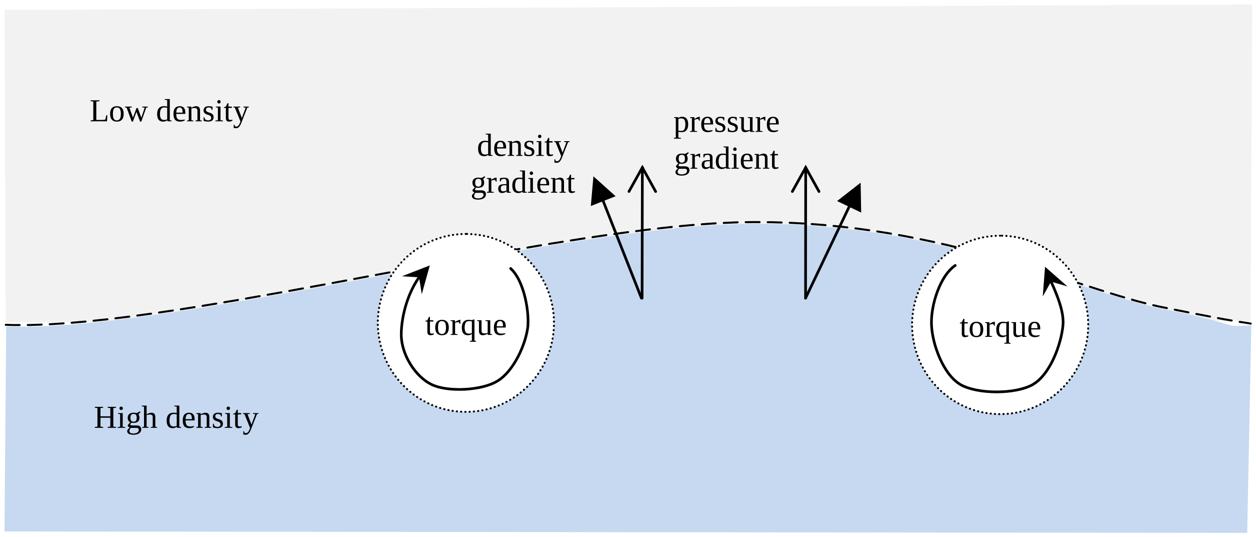


Figure 1. Making fluid right-side up. Where the pressure gradient and density gradient are not parallel, vorticity forms to bring fluid into equilibrium. Fluid will try to rotate to push down the bulge.

Boussinesq Approximation

Notice that the buoyancy term includes a density gradient ($\vec{\nabla}\rho$) and a pressure gradient ($\vec{\nabla}p$). The *Boussinesq* approximation assumes (among other things) that the fluid is in *hydrostatic equilibrium*—that is, that the pressure gradient force balances compression because of gravity, so $\vec{\nabla}p = -\rho\vec{g}$:

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

This equation obviates the need to compute pressure or its gradients.

Note: The Boussinesq approximation assumes that density variations are small (compared to the ambient density). Furthermore, liquids, which are far more incompressible than gasses, have different density variations, but this simulation does not make that distinction. So, these simulations might violate these assumptions, in which case the simulation results depart even farther from scientific rigor. For visual effects in video games, though, where performance is king, we care about plausibility and aesthetics; so once again, we take drastic liberties and favor speed over accuracy.

Adding Fluid Buoyancy to the Simulation

To add the effects of fluid buoyancy, the simulation needs to track fluid density so that it can compute its gradient. From the previous version of the simulation, particles already have density, which is used to interact with solid bodies (as described in Part 4). The simulation reuses that density to populate a grid, and then computes density on that grid in much the same way vorticity is transferred to a grid to compute its spatial derivatives.

Representing Density Using Particles

As described in earlier articles, vorticity advects as a material property; vortex particles move in the same way mass does. So, the density associated with vortex particles suffices to provide all the density information you need.

For efficiency, use as few vortons as possible—just enough to make the simulation "look good." The grid will have cells that contain no vortons. But the fluid has mass everywhere, not just at or near vortons. So, you could write the simulation so that it ensures it has at least one vorton per cell, spontaneously creating vortons in any empty cells. But that would be expensive.

Instead, you can represent density in two parts: ambient and deviation. The *ambient density* acts like a default value for density in the absence of vortons. Vortons, meanwhile, represent *density deviation* from that ambient. The actual density is then ambient plus deviation. Consider, for example, a fluid with some uniform ambient density (as the gray area in Figure 2). Then introduce vortons that have a positive density deviation (as the blue vorton in Figure 2). Near those heavier vortons, the density of the fluid is *above* ambient. Now introduce vortons that have a negative density deviation (as the pink vorton in Figure 2). Near those lighter vortons, the density of the fluid is *below* ambient.

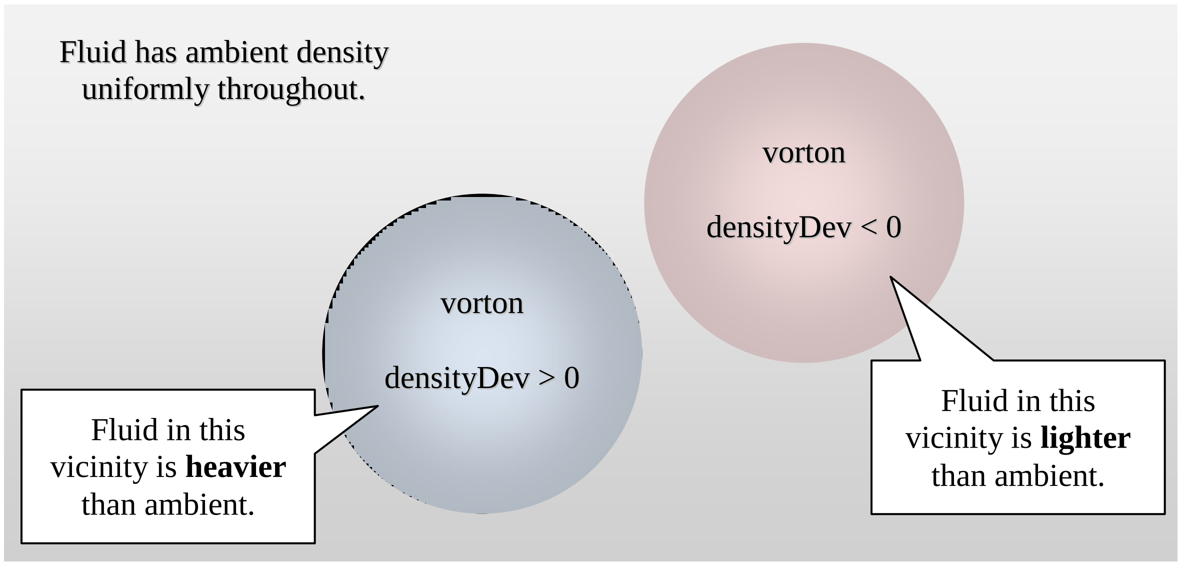

Fluid has ambient density uniformly throughout.
vorton
densityDev < 0
vorton
densityDev > 0
Fluid in this vicinity is **heavier** than ambient.
Fluid in this vicinity is **lighter** than ambient.

Figure 2. Density as ambient plus deviation. Fluid (the gray field) has an ambient density uniformly throughout. Vortons carry a value used as the deviation applied to that ambient value. The reddish vorton has a negative density deviation; therefore, fluid in that region is lighter than ambient. The bluish vorton has a positive density deviation; therefore, fluid in that region is heavier than ambient.

The code accompanying this article has a utility routine—`Particle::GetMass`—for computing the actual mass of a fluid particle, given the ambient density of the fluid, assuming that the particle density actually represents density deviation. The fluid–body collision routines from Part 4, which assumed that particle mass is their entire mass, have been changed for the present article to use the new approach of ambient plus deviation.

```
float Particle::GetMass( float ambientDensity ) const
{
    const float fMass = ( mDensity + ambientDensity ) * GetVolume() ;
    return fMass ;
}
```

Assigning Density to a Grid

As with vorticity, representing density to a grid simplifies computing spatial derivatives. So, you assign density to a `UniformGrid`. Some fluid techniques represent mass directly on a grid, not using particles. But, as Part 4 describes, that approach leads to artificially large numerical diffusion of mass—mass gets “blurry” and loses detail. In contrast, this simulation tracks mass with particles, avoiding that numerical diffusion; this particle-based simulation therefore retains more detail than grid-based simulations.

Remember from Part 6 that when assigning vorticity to a grid, the total circulation that vortons carry needs to match total circulation in the grid. Likewise, the total mass that the vortons carry needs to match total mass in the grid. When assigning mass from vorton to grid, you therefore multiply by the ratio of volumes of the vorton and grid cell, as shown in Figure 3.

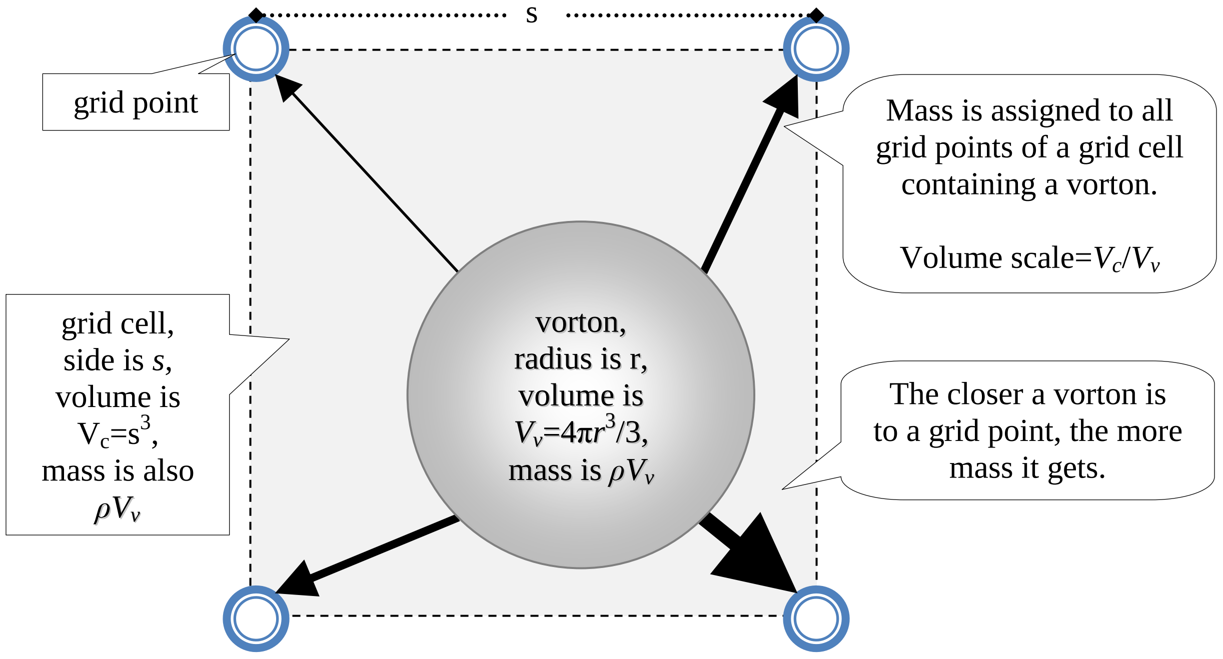

s
grid point
Mass is assigned to all grid points of a grid cell containing a vorton.
Volume scale=V_c/V_v
grid cell, side is s, volume is $V_c=s^3$, mass is also ρV_v
vorton, radius is r, volume is $V_v=4\pi r^3/3$, mass is ρV_v
The closer a vorton is to a grid point, the more mass it gets.

Figure 3. Assigning mass from vortons to grid

The contribution of mass to each grid point depends on the proximity of the vorton to that grid point; grid points closer to a vorton get more mass from it. (As in Part 6, any grid cell can contain any number of vortices: 0, 1 or more. Also, as before, the grid automatically sizes itself each update to fit all particles.) In the code, `UniformGrid::Accumulate` applies that formula.

In the code, `VortonSim::PopulateDensityGrid` assigns density from vortons to a grid:

```
void VortonSim::PopulateDensityGrid( UniformGrid<  float    > & densityGrid
                                   , const Vector< Particle > & particles  )
{
    densityGrid.Clear() ;
    densityGrid.CopyShape( mGridTemplate ) ;
    densityGrid.Init( 0.0f )               ;
    // The grid resulting from this operation must preserve the
    // same total mass that the particles represent, so multiply
    // by the ratio of the particle volume to gridcell volume.
    const float oneOverCellVolume   = 1.0f / densityGrid.GetCellVolume() ;
    // Populate density grid.
    const size_t numParticles = particles.Size() ;
    for( size_t uParticle = 0 ; uParticle < numParticles ; ++ uParticle )
    {   // For each particle in the array...
        const Particle  &   rParticle   = particles[ uParticle ] ;
        const Vec3      &   rPosition   = rParticle.mPosition   ;
        const unsigned      uOffset     = densityGrid.OffsetOfPosition( rPosition ) ;
        // Note that the formula below uses Particle::GetMass
        // instead of Particle::GetMass( ambientDensity ).
        // This is because we first want to accumulate the
        // deviation from ambient only.  The uniform ambient density
        // is irrelevant to the gradient anyway.  Also, this loop
        // only accumulates a quantity where particles exist,
        // but in the absence of particles, the fluid has mass anyway.
        const float volumeCorrectedDensity = rParticle.GetMass() * oneOverCellVolume ;
        densityGrid.Accumulate( rPosition , volumeCorrectedDensity ) ;
        // Likewise here we only want to accumulate the mass /deviations/ of particles,
        // since this will be compared later to the mass /deviations/ on the grid.
    }
}
```

Computing Density Gradients

Computing baroclinic generation requires knowing density gradients, $\vec{\nabla}\rho$, at the location of each vorton. Because it is uniform, the ambient density does not contribute to the gradient, so omit it when assigning density to the grid.

The `UniformGrid::ComputeGradient` function wraps the worker routine, `ComputeGradientInteriorSlice`, which computes the gradient for a portion of the interior of the domain. Boundaries require special logic, and they're relatively small compared to the interior, so you needn't bother parallelizing them.

```
void ComputeGradient(        UniformGrid< Vec3   > & gradient
                     , const UniformGrid< float > & scalarVals )
{
    ...
    const size_t    dims[3] = { scalarVals.GetNumPoints( 0 )
                              , scalarVals.GetNumPoints( 1 )
                              , scalarVals.GetNumPoints( 2 )    } ;
    const size_t    dimsMinus1[3] = { scalarVals.GetNumPoints( 0 )-1
                                    , scalarVals.GetNumPoints( 1 )-1
                                    , scalarVals.GetNumPoints( 2 )-1 } ;

    // Compute derivatives for interior (i.e. away from boundaries).
    #if USE_TBB
        {   // Use Threading Building Blocks to parallelize the computation.
            const size_t numZ = dimsMinus1[2] - 1 ;
            // Estimate grain size.
            const size_t grainSize =  MAX2( 1 , numZ / gNumberOfProcessors ) ;
            parallel_for(
                tbb::blocked_range<size_t>( 1 , dimsMinus1[2] , grainSize )
        , UniformGrid_ComputeGradientInterior_TBB( gradient , scalarVals ) ) ;
        }
    #else
        ComputeGradientInteriorSlice(gradient,scalarVals,1,dimsMinus1[2]);
    #endif
    // Compute derivatives for boundaries: 6 faces of box.
    ...
}
```

The actual computation occurs inside `ComputeGradientInteriorSlice`. The “slice” approach facilitates parallelizing using Intel® Threading Building Blocks (Intel® TBB).

```
static void ComputeGradientInteriorSlice(       UniformGrid< Vec3  > & gradient
                                        , const UniformGrid< float > & val
                                        , size_t izStart , size_t izEnd )
{
    const Vec3      spacing                  = val.GetCellSpacing() ;
    // Avoid divide-by-zero when z size is effectively 0 (for 2D domains)
    const Vec3      reciprocalSpacing( 1.0f / spacing.x , 1.0f / spacing.y
                        , spacing.z > FLT_EPSILON ? 1.0f / spacing.z : 0.0f ) ;
    const Vec3      halfReciprocalSpacing( 0.5f * reciprocalSpacing ) ;
    const size_t    dims[3] = { val.GetNumPoints( 0 ) , val.GetNumPoints( 1 )
                               , val.GetNumPoints( 2 )    } ;
    const size_t    dimsMinus1[3] = { val.GetNumPoints( 0 )-1
                        , val.GetNumPoints( 1 )-1 , val.GetNumPoints( 2 )-1 } ;
    const size_t    numXY                    = dims[0] * dims[1] ;
    size_t          index[3] ;

    // Compute derivatives for interior (i.e. away from boundaries).
    for( index[2] = izStart ; index[2] < izEnd ; ++ index[2] )
    {
        ASSIGN_Z_OFFSETS ;
        for( index[1] = 1 ; index[1] < dimsMinus1[1] ; ++ index[1] )
        {
            ASSIGN_YZ_OFFSETS ;
            for( index[0] = 1 ; index[0] < dimsMinus1[0] ; ++ index[0] )
            {
                ASSIGN_XYZ_OFFSETS ;

                Vec3 & rVector = gradient[ offsetX0Y0Z0 ] ;
                /* Compute d/dx */
                rVector.x = ( val[ offsetXPY0Z0 ] - val[ offsetXMY0Z0 ] )
                            * halfReciprocalSpacing.x ;
                /* Compute d/dy */
                rVector.y = ( val[ offsetX0YPZ0 ] - val[ offsetX0YMZ0 ] )
                            * halfReciprocalSpacing.y ;
                /* Compute d/dz */
                rVector.z = ( val[ offsetX0Y0ZP ] - val[ offsetX0Y0ZM ] )
                            * halfReciprocalSpacing.z ;
            }
        }
    }
}
```

Compute the Baroclinic Generation of Vorticity

You again use Intel® TBB to parallelize computing the baroclinic term:

```
void VortonSim::GenerateBaroclinicVorticity( const float    & timeStep
                                             , const unsigned & uFrame  )
{
    if( fabsf( mGravAccel * mVelGrid.GetExtent() ) < FLT_EPSILON )
    {   // Domain is 2D and has no significant component along the gravity direction,
        // so baroclinic generation cannot occur in this Boussinesq approximation.
        return ;
    }

    // Populate density grid.
    PopulateDensityGrid( mDensityGrid
                       , reinterpret_cast< Vector< Particle > & >( mVortons ) ) ;

    // Compute density gradient.
    mDensityGradientGrid.Clear() ; // Clear any stale velocity information
    mDensityGradientGrid.CopyShape( mGridTemplate ) ;
    mDensityGradientGrid.Init() ; // Reserve memory for density gradient grid.
    ComputeGradient( mDensityGradientGrid , mDensityGrid ) ;

    const size_t    numVortons      = mVortons.Size() ;

    #if USE_TBB
        // Estimate grain size based on size of problem and number of processors.
        const size_t grainSize =  MAX2( 1 , numVortons / gNumberOfProcessors ) ;
        // Fill vertex buffer using threading building blocks
        parallel_for( tbb::blocked_range<size_t>( 0 , numVortons , grainSize )
                    , VortonSim_GenerateBaroclinicVorticity_TBB( timeStep , this ) ) ;
    #else
        GenerateBaroclinicVorticitySlice( timeStep , 0 , numVortons ) ;
    #endif
}
```

The `VortonSim::GenerateBaroclinicVorticitySlice` routine computes the baroclinic term of the vorticity equation applied to each vorton. This step transfers back from grid to particles, completing the cycle:

```
void VortonSim::GenerateBaroclinicVorticitySlice( float timeStep
                                                , size_t iPclStart , size_t iPclEnd )
{
    for( size_t iPcl = iPclStart ; iPcl < iPclEnd ; ++ iPcl )
    {   // For each particle in this slice...
        Vorton &    rVorton         = mVortons[ iPcl ] ;
        // Technically the baroclinic term should have density in the denominator,
        // but it's relatively expensive to calculate the local value, plus the
        // Boussinesq approximation assumes small density variations.
        // Also, when density variations are large, the interpolation
        // sometimes leads to non-physical artifacts like negative densities.
        // So instead, we use the ambient density in the denominator,
        // which yields the correct units and gets us in the right ballpark.
        // This is yet another example of where, for visual effects,
        // we take drastic liberties in the name of speed over accuracy.
        Vec3        densityGradient ;
        mDensityGradientGrid.Interpolate( densityGradient , rVorton.mPosition ) ;
        const Vec3 baroclinicGeneration = densityGradient ^ mGravAccel / mAmbientDensity;
        // The formula for line below is meant to update vorticity, but since
        // we store that as angular velocity, formula has an extra factor of 0.5.
        rVorton.mAngularVelocity += 0.5f * baroclinicGeneration * timeStep ;
    }
}
```

Results

Figure 4 shows a simulation with an initially motionless ball of heavy fluid, colored blue. The resulting motion looks like dye dropped into a glass of water and sinking, vaguely resembling the results of a Rayleigh–Taylor (RT) instability.

Note: The classic RT instability has (initially) nearly flat layers of fluid with different densities, which are in an unstable equilibrium. This droplet example is not really an example of the RT instability but the underlying cause of the RT instability is the same: fluid buoyancy.

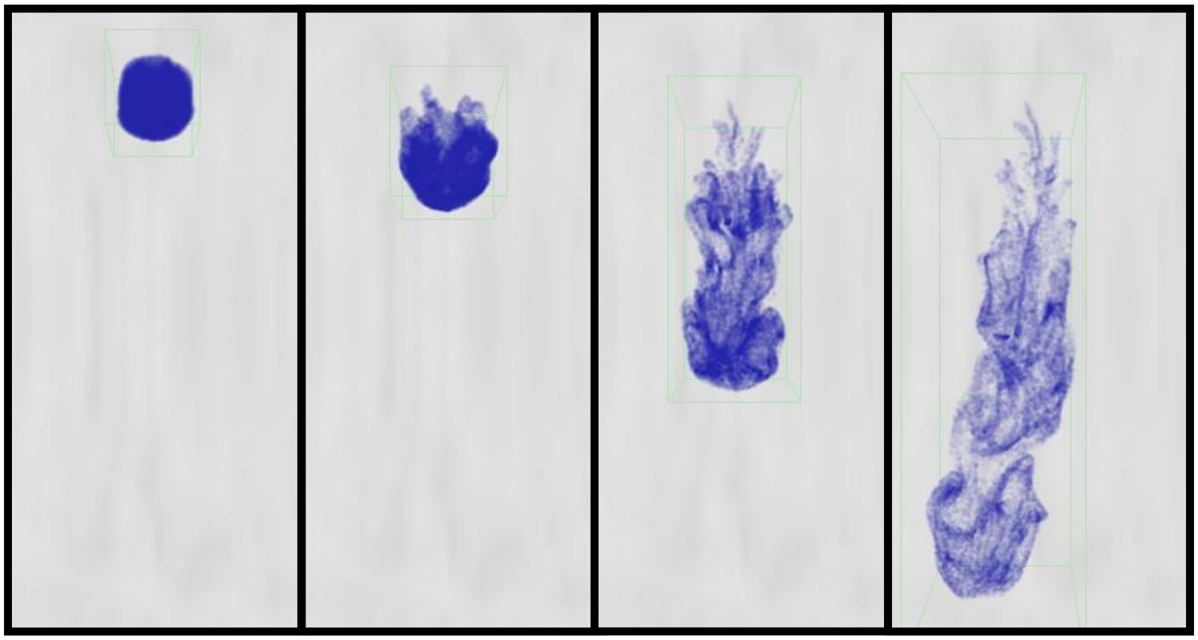


Figure 4. Dye drop falling through fluid. Initially, the ball of fluid, which is heavier than the fluid around it, has no velocity. Then, it begins to fall and forms a vortex ring, leaving behind tendrils of dye.

Figure 5 shows two drops of fluid—one heavier (blue) and one lighter (red) than the surrounding fluid. The drops are initially motionless but soon start to move as a result of buoyancy. They form a pair of vortex rings that eventually collide.

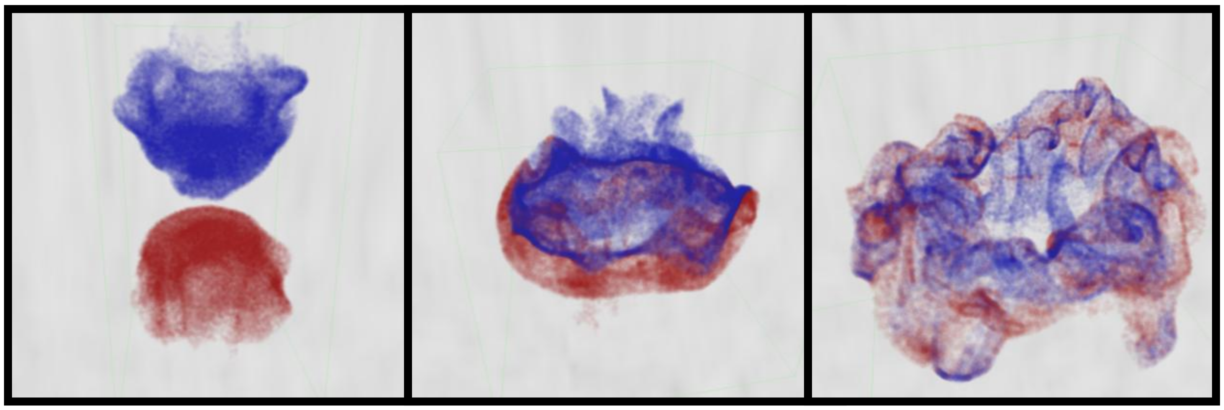


Figure 5. Dye drops falling and rising through fluid. The heavier blue drop falls, the lighter red drop rises, and then they collide.

Performance

Performance profiles reveal that the fluid buoyancy computation, including particle-to-grid transfer, density gradient, and the baroclinic term calculation, uses only around 0.3 percent of the total time—three one-thousandths—which is nearly negligible. Typical durations for that computation, for these simulations, ran for only tens of microseconds per frame on even modest hardware like an Intel® Pentium® 4 processor running at 3.0 GHz, with significantly faster performance on an Intel® Core™2 Duo processor running at 2.6 GHz.

Coming Up

Future articles will build upon this installment, adding solid-body buoyancy, convection (thermal expansion and diffusion), and combustion (generating heat by chemical processes). These articles will combine for even more interesting and interactive effects, allowing game effects creators to let their users interact with their virtual worlds with even greater immersion.

Further Reading

- Koumoutsakos, Cottet, & Rossinelli. (2008). Flow simulations using particles. (Note that their equation 67 has the wrong units for the baroclinic term, missing a ρ in the denominator. They are likely assuming very small density variation and that the ambient density is 1.0 in whatever units their simulation uses.)