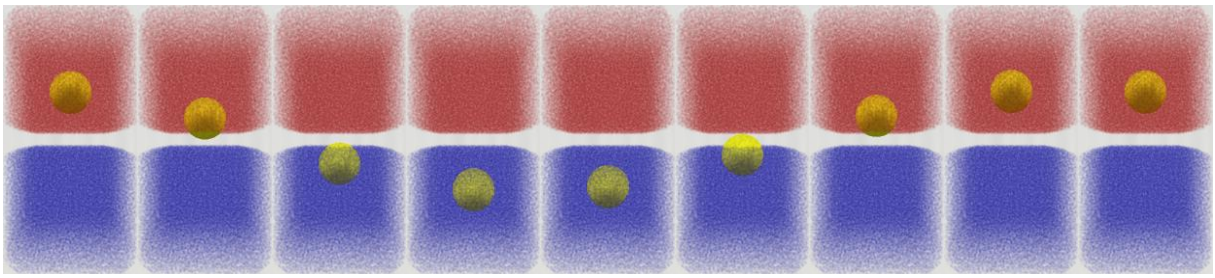
Fluid Simulation for Video Games (part 9)

By Dr. Michael J. Gourlay



Archimedes' Principle: Body Buoyancy

Any object wholly or partially immersed in a fluid is buoyed up by a force equal to the weight of the fluid displaced by the object. This is *Archimedes' Principle.*

This article—the ninth in a series—describes how to approximate buoyant and gravitational forces on a body immersed in a fluid with varying density. 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. Part 8 explained how a vortex-based fluid simulation can handle variable density in a fluid.

This article introduces features to simulation code presented in previous articles: Now, bodies immersed within the fluid float or sink depending on the mass of fluid the body displaces. This new feature augments how visual effects have a two-way interaction with physical objects in the simulation.

The Weight of Fluid Displaced by an Object

How do you express Archimedes' principle in a form usable by a simulation?

This formula expresses the net force acting on a body immersed within a fluid with constant density:

$$\underset{\text{Net Force}}{F} = \underset{\text{Body weight}}{m_{body} g} - \underset{\text{Body buoyancy}}{\rho_{fluid} V_{body} g}$$

Here, m_{body} is the mass of the body, g is the acceleration caused by gravity, ρ_{fluid} is the density of the fluid, and V_{body} is the volume of the body.

If the body is only partially submerged, then V_{body} is the volume of the *portion* of the body submerged, as Figure 1 shows.

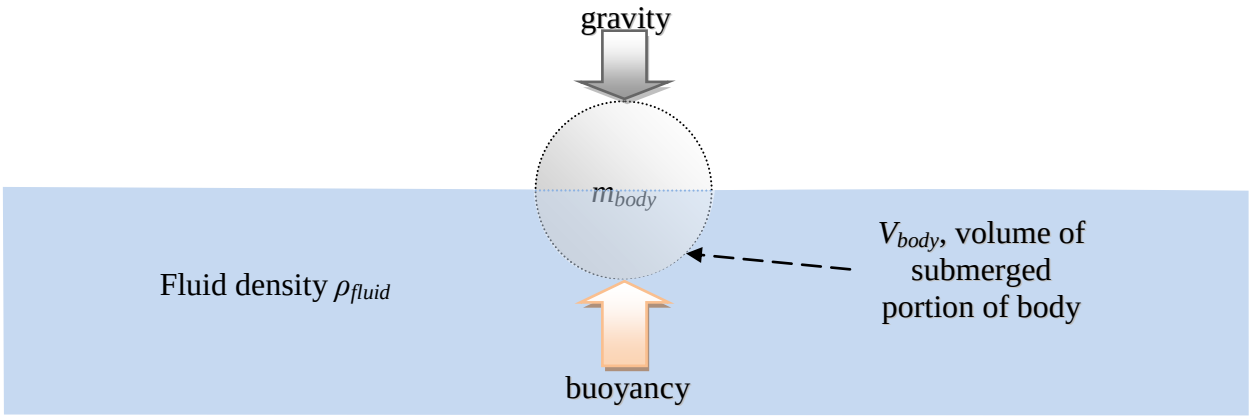


Figure 1. *Buoyancy of a partially submerged body*

Mass of Fluid Displaced for a Fluid with Variable Density

Although the formula above works fine for a body *fully* submerged in a fluid with *constant* density, it needs modification for a body either *partially* submerged or (equivalently) submerged in a fluid with *varying* density. In that case, you effectively need to subdivide the fluid mass displaced into multiple terms as follows:

$$\underset{\text{Fluid mass displaced}}{m_{displaced}} = \underset{\text{portion 1}}{\rho_{fluid,1}V_{body,1}} + \underset{\text{portion 2}}{\rho_{fluid,2}V_{body,2}} + \ldots$$

In the limit of continuous density variation, the sum above would become an integral. But a computer simulation would have to discretize that integral, so leave it as a sum. Figure 2 depicts a case in which the fluid has two different density values.

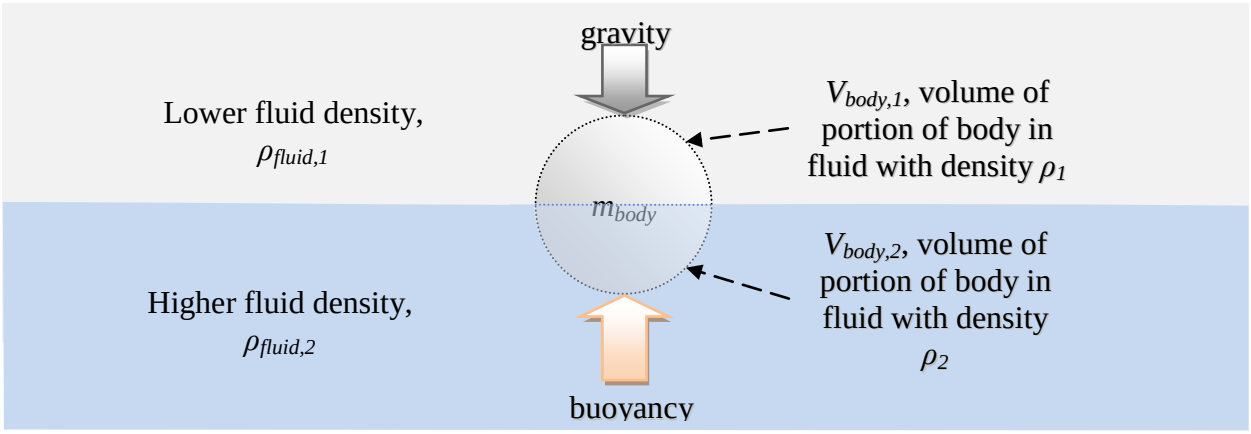


Figure 2. *Buoyancy of a body immersed in a fluid with two different densities*

Adding Body Buoyancy to the Simulation

To add the effects of body buoyancy, the simulation must compute gravity and buoyancy forces for each body. From Part 7, the simulation code already has density, which is used to compute fluid buoyancy. The augmented simulation presented in this article reuses that density grid to approximate the mass of fluid displaced by the body.

The simulation code requires these modifications:

- Add a method named `BuoyBodies` to `FluidBodySim`.
- Call `FluidBodySim::BuoyBodies` from `PclOpFluidBodyInteraction::Operate`.

For simplicity, the code for `FluidBodySim::BuoyBodies` samples fluid density at a fixed number of locations within the body region and uses the average of those samples as the average density of the fluid displaced by the body. It also treats the body as though the volume of body associated with each sample were the same.

Note: *This is a drastic simplification and can be made more sophisticated if required, but this article focuses on visual effects and therefore assumes that the rigid body simulation needs only a modest amount of impact from the particle system. Typically, if you want more sophisticated rigid body physics, you would use a proper physics engine to compute forces. In contrast, this technique is meant to tie results from the particle system into an existing physics engine by computing a quick-and-dirty approximation of buoyancy and feeding that force to the physics engine via a routine like* `ApplyBodyForce`.

```
void FluidBodySim::BuoyBodies( Vector< Particle > & particles
    , const UniformGrid< float > & densityDeviationGrid , float ambientFluidDensity
    , const Vec3 & gravityAcceleration , const Vector< RigidBody * > & rigidBodies )
{
    const size_t    numBodies       = rigidBodies.Size() ;
    const size_t    numParticles    =   particles.Size() ;
    const Vec3      gravityDir      = gravityAcceleration.Direction() ;

    for( unsigned uBody = 0 ; uBody < numBodies ; ++ uBody )
    {   // For each body in the simulation...
        RbSphere &  rSphere         = (RbSphere &) * rigidBodies[ uBody ] ;
        // Compute profile of fluid density around body
        float densityDeviationAtQueryPoint   ; // fluid density at query points.
        float densityDeviationSum        ; // Average fluid density in region of body.
        float divisor            = 1.0f ;
        // Sample fluid density at multiple places within the body region.
        Vec3 vQueryPos = rSphere.mPosition ;
        if( densityDeviationGrid.Encompasses( vQueryPos ) )
        {
            densityDeviationGrid.Interpolate( densityDeviationSum , vQueryPos ) ;
        }
        vQueryPos = rSphere.mPosition + rSphere.mRadius * gravityDir ;
        if( densityDeviationGrid.Encompasses( vQueryPos ) )
        {
            densityDeviationGrid.Interpolate( densityDeviationAtQueryPoint , vQueryPos ) ;
            densityDeviationSum += densityDeviationAtQueryPoint ;
            divisor += 1.0f ;
        }
        vQueryPos = rSphere.mPosition - rSphere.mRadius * gravityDir ;
        if( densityDeviationGrid.Encompasses( vQueryPos ) )
        {
            densityDeviationGrid.Interpolate( densityDeviationAtQueryPoint , vQueryPos ) ;
            densityDeviationSum += densityDeviationAtQueryPoint ;
            divisor += 1.0f ;
        }
        // Average fluid density samples.
        const float densityAverage  = densityDeviationSum / divisor + ambientFluidDensity ;
        // Approximate body buoyancy force.
        const float massDisplaced   = densityAverage * rSphere.GetVolume() ;
        const float bodyMass        = 1.0f / rSphere.GetInverseMass() ;
        // Sum buoyancy and gravity forces.
        const Vec3  netForce        = gravityAcceleration * ( bodyMass - massDisplaced ) ;
        rSphere.ApplyBodyForce( netForce ) ;
    }
}
```

Speeding Up the Simulation

Computing body buoyancy costs some CPU time (see below to see exactly how much). To recover that cost, use Intel® Threading Building Blocks (Intel® TBB) to parallelize one of the other computations.

According to performance profiles, the slowest serial process is the one that computes diffusion of vorticity, `DiffuseVortonPSE`. Part of that process is embarrassingly parallel, because each vorton modified only needs Read access to the vortons in its neighborhood.

The following steps lead to a parallelized diffusion calculation:

1. Make a new routine called `DiffuseVorticityPSESlice`.
2. Create a new class called `VortonSim_DiffuseVorticityPSE_TBB`.
3. Change `DiffuseVorticityPSE` to use the refactored code.

Parallelized Vorticity Diffusion Code

Extract the `for` loops from `DiffuseVorticityPSE` into a new routine called `DiffuseVorticityPSESlice`. This routine takes start and end indices supplied by Intel TBB.

```
void VortonSim::DiffuseVorticityPSESlice( const float & timeStep
    , const UniformGrid< Vector< unsigned > > & ugVortIdx , size_t izStart , size_t izEnd )
{
    // Exchange vorticity with nearest neighbors

    const size_t   & nx     = ugVortIdx.GetNumPoints( 0 ) ;
    const size_t     nxm1   = nx - 1 ;
    const size_t   & ny     = ugVortIdx.GetNumPoints( 1 ) ;
    const size_t     nym1   = ny - 1 ;
    const size_t     nxy    = nx * ny ;
    const size_t   & nz     = ugVortIdx.GetNumPoints( 2 ) ;
    const size_t     nzm1   = nz - 1 ;

    size_t idx[3] ;
    for( idx[2] = izStart ; idx[2] < izEnd ; ++ idx[2] )
    {   // For all points along z within a region...
        ...
    }
}
```

The new class, `VortonSim_DiffuseVorticityPSE_TBB`, which calls `VortonSim::DiffuseVorticityPSESlice`, takes the form of a function, as Intel TBB requires:

```
class VortonSim_DiffuseVorticityPSE_TBB
{
    float                                       mTimeStep   ;   ///< Duration of time step
    VortonSim *                                 mVortonSim  ;   ///< VortonSim object
    const UniformGrid< Vector< unsigned > > &   mUgVortIdx  ;   ///< Grid of vorton indices
    public:
      void operator() ( const tbb::blocked_range<size_t> & r ) const
      {   // Compute subset of vorticity diffusion.
        mVortonSim->DiffuseVorticityPSESlice( mTimeStep , mUgVortIdx , r.begin() , r.end() );
      }
        VortonSim_DiffuseVorticityPSE_TBB( float timeStep , VortonSim * pVortonSim
                                    , const UniformGrid< Vector< unsigned > > & ugVortIdx )
            : mTimeStep( timeStep )
            , mVortonSim( pVortonSim )
            , mUgVortIdx( ugVortIdx )
        {}
} ;
```

Change `DiffuseVorticityPSE` to use the refactored code, via Intel TBB's `parallel_for`:

```
void VortonSim::DiffuseVorticityPSE( const float & timeStep , const unsigned & uFrame )
{
    // Phase 1: Partition vortons

    // Create a spatial partition for the vortons.
    // Each cell contains a dynamic array of integers
    // whose values are offsets into mVortons.
    UniformGrid< Vector< unsigned > > ugVortIdx( mGridTemplate ) ;
    ugVortIdx.Init() ;

    const size_t numVortons = mVortons.Size() ;

    for( unsigned offset = 0 /* Start at 0th vorton */ ; offset < numVortons ; ++ offset )
    {   // For each vorton...
        Vorton &    rVorton         = mVortons[ offset ] ;
        // Insert the vorton's offset into the spatial partition.
        ugVortIdx[ rVorton.mPosition ].PushBack( offset ) ;
    }

    // Phase 2: Exchange vorticity with nearest neighbors

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

    // Estimate grain size based on size of problem and number of processors.
    const size_t grainSize =  MAX2( 1 , nzm1 / gNumberOfProcessors ) ;
    // Compute vorticity diffusion using threading building blocks
    parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize )
        , VortonSim_DiffuseVorticityPSE_TBB( timeStep , this , ugVortIdx ) ) ;
}
```

Results

Figure 3 shows a simulation with a ball whose density lies between that of the light fluid (at the top, colored red) and the heavier fluid (at the bottom, colored blue).

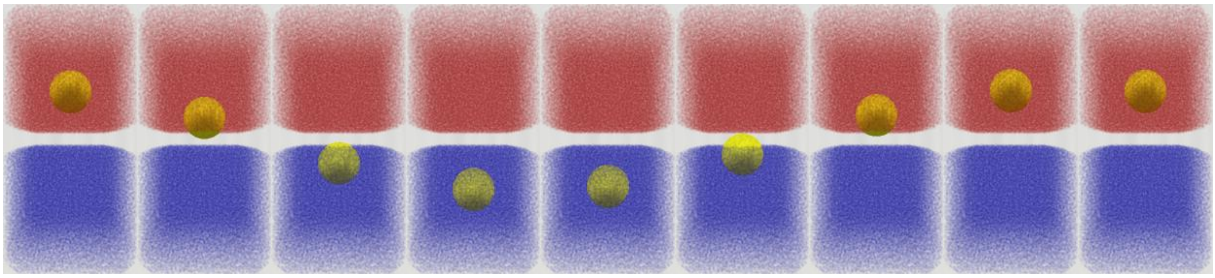


Figure 3. *Ball buoys in a density-stratified fluid*

Performance

Performance profiles reveal that the body buoyancy computation uses well under 1 percent of the total time. Typical durations for that computation, for these simulations, ran for only 2 microseconds per frame on even modest hardware like a computer running 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.

Table 1 shows durations for various processes while running the simulation on an Intel® Xeon® processor X5660 running at 2.8 GHz.

Table 1. Run Durations for Processes on an Intel® Xeon® processor

Threads	Buoy (ms)	Diffuse (ms)	Total (ms)
1	0.000733	0.323733	33.6
2	0.000733	0.318233	19.0
3	0.000733	0.287	12.7
4	0.000733	0.257373	11.9
5	0.000733	0.218144	11.9
6	0.000733	0.207145	11.4

Coming Up

Future articles will build on this installment, adding *convection* (thermal expansion and diffusion) and *combustion* (generating heat by chemical processes). These additions will give the code the ability to simulate smoldering and burning.