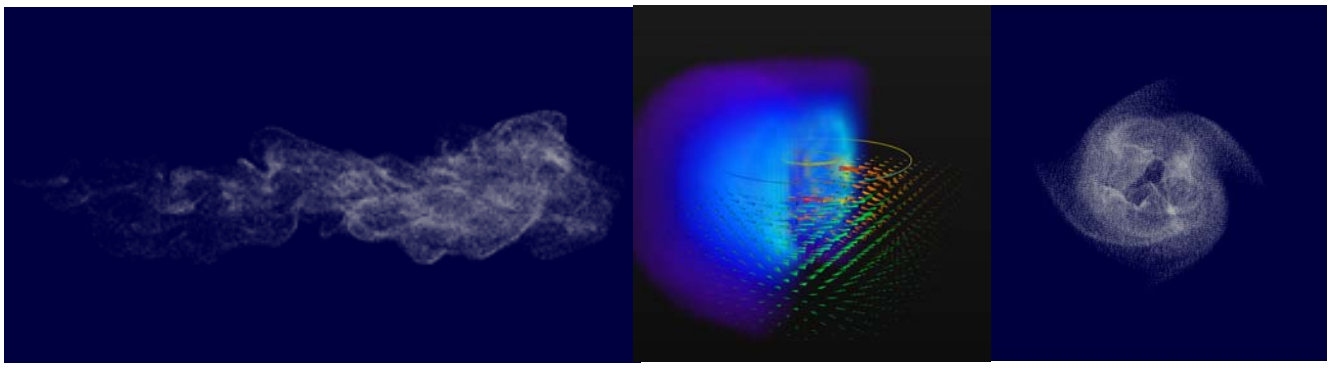
Fluid Simulation for Video Games (part 4)

By Dr. Michael J. Gourlay



Two-way Fluid-Body Interaction

This article, the fourth in a series, augments a fluid simulation, presented in the third article, to include two-way fluid-body interaction. The first article summarized fluid dynamics, the second surveyed fluid simulation techniques and the third presented a vortex-particle fluid simulation that runs in real time. The additions presented in this article allow the flow to interact with rigid bodies and vice versa.

This simulation also exploits the embarrassingly parallel nature of the algorithms and uses Intel® Threading Building Blocks (TBB) to spread the work across multiple threads.

Source code accompanies this article, which demonstrates the concepts behind the simulation but does not provide an explanation of every line of code. The code contains plenty of comments, so please read it for further elucidation.

After this article, this series continues with more advanced topics, including performance analysis, optimization, alternative algorithms, and more sophisticated parallelism. Future articles will also grant you creative license to deviate even further from the rigors of math and science, because the goal here is to make compelling, interactive virtual worlds, not research fluid dynamics.

The simulation handles boundary conditions and two-way interactions between fluid and rigid bodies. You also get to see some demonstrations of rigid bodies interacting with the fluid. The article concludes with a roadmap of future articles on how to enhance the simulation further—both in performance and in functionality—and how to incorporate this subsystem into a game engine.

Boundary Conditions

As mentioned in the previous article, this simulation behaves as though the computational domain has no limits, in the sense that fluid is allowed to flow outward indefinitely. These conditions are called *open flow* boundary conditions, and they arise from the fact that, at the start of each frame, the uniform grids used to compute the influence tree and the velocity grid are both sized to contain all particles (vortons and tracers) in the simulation.

Although the idea of assigning vorticity to satisfy velocity boundary conditions is not new, this particular treatment *is* new to the world. I formulated it to be simple and fast, even at the expense of accuracy. And if you look at the code, you will find that it contains multiple variations on this treatment, so feel free to experiment. As the results below show, this approach serves its intended purpose well. This is the mantra of visual effects for video games: Look pretty, run fast.

The fluid also interacts with rigid bodies, in which case the flow satisfies no-slip and no-through boundary conditions. The `FluidBodySim::SolveBoundaryConditions` method implements this by reassigning vortex particles that become embedded within a rigid body. First, the simulation ejects particles outside of the rigid body. Next (as shown in Figure 1) the simulation reassigns vorticity such that the flow velocity satisfies boundary conditions at the contact point, which is approximately the point where the particle collided with the body.

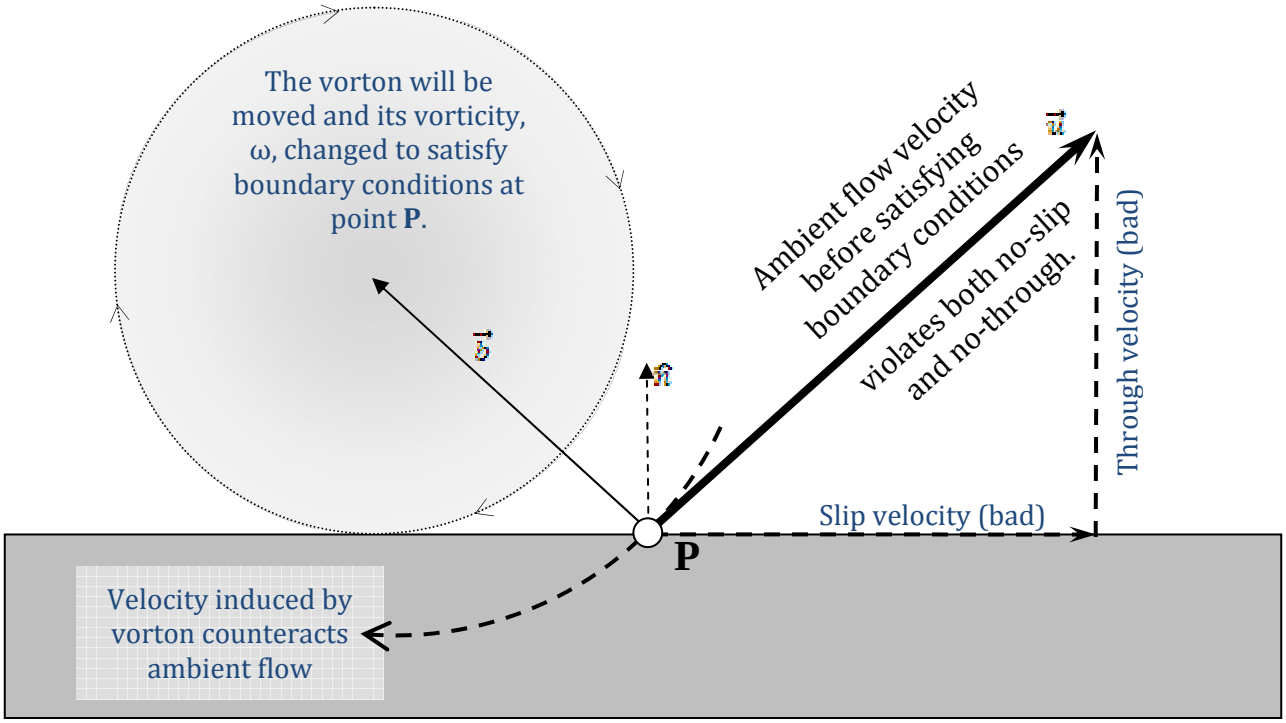


Figure 1. Assigning a vorton to satisfy velocity boundary conditions on the surface of a rigid body

Assigning vorticity to satisfy the boundary conditions on velocity requires three parts:

- Knowing the desired velocity induced by the vorton at the contact point
- Knowing where to place the vorton
- Knowing what to assign its vorticity

Knowing the velocity the vorton should induce seems like a simple problem: The flow velocity (relative to the body) needs to be zero at the contact point (point *P* in Figure 7), which is on the body surface. But the flow velocity prior to satisfying the boundary condition includes contributions due to all vortons in the flow, including all vortons about to be moved. So, this ends up being a complicated problem. But you can simplify it by treating one vorton at a time and assuming that all others remain unchanged. You can readily obtain the pre-collision fluid flow velocity from the velocity grid, as was done in the advection phase. From that, subtract the velocity due to the vorton (as provided by `Vorton::AccumulateVelocity`) about to be moved. Doing so gives an estimate for the velocity $\vec{u}$ at the contact point to the ambient flow field.

Next, place the vorton. It might seem reasonable to place the vorton outside the body surface, above the contact point, along the direction of the surface normal, $\hat{n}$. But fluid flows in a circle around the vorton. If the vorton is directly above the contact point, then that vorton would only induce velocity tangential to the surface there. Then, the vorton

could satisfy the no-slip condition but not the no-through condition. So, the vorton needs to be offset from the surface normal along some vector $\vec{b}$ such that the direction tangent to the vorton exactly counters the ambient flow direction $\vec{u}$ at P. You also know that $\vec{\omega}$ must be perpendicular to both $\hat{u}$ and $\hat{n}$, hence $\hat{\omega} = \hat{u} \times \hat{n}$. This condition tells you that $\hat{b} = \hat{\omega} \times \hat{u}$, which in turn tells you the direction of $\vec{b}$, so $\hat{b} = \hat{u} \times \hat{n} \times \hat{u}$. You have freedom to choose its length, so choose $|\vec{b}| = \sigma$, the radius of the vorton.

Finally, assign vorticity $\vec{\omega}$ to the vorton such that the velocity it induces at *P* exactly opposes $\vec{u}$. Simply rewrite the formula that gives velocity in terms of vorticity, so that it gives vorticity as a function of velocity: $\omega = \frac{4\pi r^2 u}{V}$. The `Vorton::AssignByVelocity` method implements this formula.

Each collision leads to a change in vorticity. To conserve angular momentum, the system must keep track of these changes—known as *impulsive torques*—and apply them somewhere else.

Rigid Body Motion

This simulation includes an extremely simplified rigid body simulation, implemented in the `RigidBody` class. It is meant to stand proxy for a "real" physics engine, to show an example of how to tie this fluid simulation to a rigid body simulation.

Each time a vorton collides with a rigid body, the simulation changes vorticity (and hence angular momentum) as described in the previous section. When the simulation changes ω, it also changes angular momentum. To conserve angular momentum, the simulation transfers that change, via an impulsive torque, to the body using `RigidBody::ApplyImpulsiveTorque`. In this way, rigid bodies mutually interact with the fluid.

Each collision between a particle and a rigid body also changes the *linear* momentum of the particle. Each change in linear momentum, an impulsive force, applied to a particle gets transferred to the rigid body (using `RigidBody::ApplyImpulse`) and thus pushes it around. This yields an effect akin to aerodynamic drag.

The collisions are inelastic, meaning that immediately after the collision, particles involved in the collision adopt the velocity of the body at the point of contact. It also means that kinetic energy is lost in the collision. In reality, that energy must go somewhere; it does not just disappear. You could account for that energy by deforming the body, raising its temperature, or raising the temperature of the fluid. But in this simple simulation, you do not track the internal state of the fluid, so effectively this energy simply dissipates like magic.

Although the results work well enough for visual effects in video games, as with other aspects of this simulation, this ad hoc treatment of fluid–body interaction deviates from rigorous treatments used in scientific research, so curious and intrepid readers should consult more sophisticated and detailed treatments of this topic, such as Cottet and Koumoutsakos (2000).

Parallelization

As in the previous article, let us again take advantage of the embarrassingly parallel nature of these operations. Where can we use Intel® TBB to parallelize the code?

The routine `AdvectTracers` has a good combination of independence and slow run times. The simulation therefore uses Intel® TBB's `parallel_for` to run it across multiple threads. As before, this routine has an associated helper routine that operates on a subset of the total number of tracer particles, and then uses a function object, which is an object that overrides `operator()`, to inform Intel® TBB how to execute the process. Because this simulation uses a large number of passive tracer particles, this turns out to be a natural fit for data parallelism.

Converting `AdvectTracers` to use Intel® TBB's `parallel_for` required minimal and trivial changes. The original routine, which encompassed a simple loop over all tracers, was modified to invoke a helper routine, `AdvectTracersSlice`. This helper routine includes the loop body, except instead of looping over all tracers, the loops start and end index values (`itStart` and `itEnd`) come from function arguments:

```
void VortonSim::AdvectTracersSlice( const float & timeStep , const unsigned & uFrame ,
                                    unsigned itStart , unsigned itEnd )
{
    for( unsigned offset = itStart ; offset < itEnd ; ++ offset )
    {   // For each passive tracer in this slice...
        Particle & rTracer = mTracers[ offset ] ;
        Vec3 velocity ;
        mVelGrid.Interpolate( velocity , rTracer.mPosition ) ;
        rTracer.mPosition += velocity * timeStep ;
        rTracer.mVelocity  = velocity ; // Cache for use in collisions
    }
}
```

At this stage, the functionality of the original serial routine simply entails calling the helper routines, passing in 0 and `numTracers` as the loop index' begin and end values:

```
void VortonSim::AdvectTracers( const float & timeStep , const unsigned & uFrame )
{
    const size_t numTracers = mTracers.Size() ;
#if USE_TBB // Parallel
    // Estimate grain size based on size of problem and number of processors.
    const size_t grainSize =  MAX2( 1 , numTracers / gNumberOfProcessors ) ;
    // Advect tracers using multiple threads.
    parallel_for( tbb::blocked_range<size_t>( 0 , numTracers , grainSize ) ,
                  VortonSim_AdvectTracers_TBB( this , timeStep , uFrame ) ) ;
#else // Serial
    AdvectTracersSlice( timeStep , uFrame , 0 , numTracers ) ;
```

```
#endif
}
```

As you can see from the code above, this version of `AdvectTracers` supports both parallel and serial versions. The parallel version invokes `parallel_for`, using a simple wrapper class called `VortonSim_AdvectTracers_TBB`:

```
class VortonSim_AdvectTracers_TBB
{       /// Functor, a.k.a. function object – a class that simply wraps a function call.
        VortonSim      * mVortonSim ;    ///< Address of VortonSim object
        const float    & mTimeStep  ;    ///< Change in virtual time since last update
        const unsigned & mFrame     ;    ///< Number of update which have occurred
    public:
        void operator() ( const tbb::blocked_range<size_t> & r ) const
        {   // Advect subset of tracers.
            mVortonSim->AdvectTracersSlice( mTimeStep , mFrame , r.begin() , r.end() ) ;
        }
        VortonSim_AdvectTracers_TBB( VortonSim * pVortonSim , const float & timeStep ,
                                     const unsigned & uFrame )
            : mVortonSim( pVortonSim )
            , mTimeStep( timeStep )
            , mFrame( uFrame )
        { /* Construct a function object */ }
} ;
```

The `VortonSim_AdvectTracers_TBB` class simply wraps the call to `AdvectTracersSlice`. Such a class, whose instances are invoked using the same syntax as a function (via overloading `operator()`) are called **functors**. Since that routine is a method of the `VortonSim` class, the functor constructor requires the address of a `VortonSim` object, and any other arguments passed to the helper routine, `AdvectTracersSlice`. Notice that `operator()` is "const", meaning that invoking the functor cannot alter the object itself. This is tantamount to saying that each thread runs entirely independent of the others, i.e. they require no communication or synchronization.

Such problems are called "embarrassingly parallel", and many aspects of physical simulations typically have this property.

Not all algorithms have this level of simplicity. For example, one of the slowest remaining processes is `FindBoundingBox`, which is almost embarrassingly parallel, but not quite. You could speed up this process in at least two ways: parallelize it, or exploit the data cache coherence by incorporating the operations into `AdvectTracers` and `AdvectVortons`. In fact, because `AdvectTracers` is already parallelized, it seems like a natural choice to put those operations inside `AdvectTracers`. Unfortunately, that is easier said than done. The operation requires writing to an address shared by all threads. Synchronizing access to the "min" and "max" variables which accumulate data across all particles would create huge contention and effectively serialize the process. Solving this contention issue requires a simple trick that a future article will explore. But these issues lie outside the scope of this article, so the next article will address them. Suffice it to say, for now, that the solution requires using another easy-to-use feature of Intel® Threading Building Blocks, called `parallel_reduce`.

Results

The previous article presented results of canonical fluid simulation situations like a self-propagating vortex ring and crossed vortex tubes. Now we introduce scenarios that include fluid-body interactions:

- A ball spinning in a fluid, which causes the fluid to swirl
- A ball passing through fluid, which creates a wake

A ball spinning inside fluid should induce the flow to swirl. In fact, it should also reproduce the velocity field inside the ball—and it does, as Figure 2 shows. This is remarkable, because the only vorticity that exists is on the surface of the ball, and that thin layer of vorticity generates a velocity field in all space—*including inside the ball itself*—and the velocity field inside the ball must be that of a solid rigid body. It seems like magic, that simply satisfying the boundary conditions "generates" so much motion in so many places – even in regions without fluid!

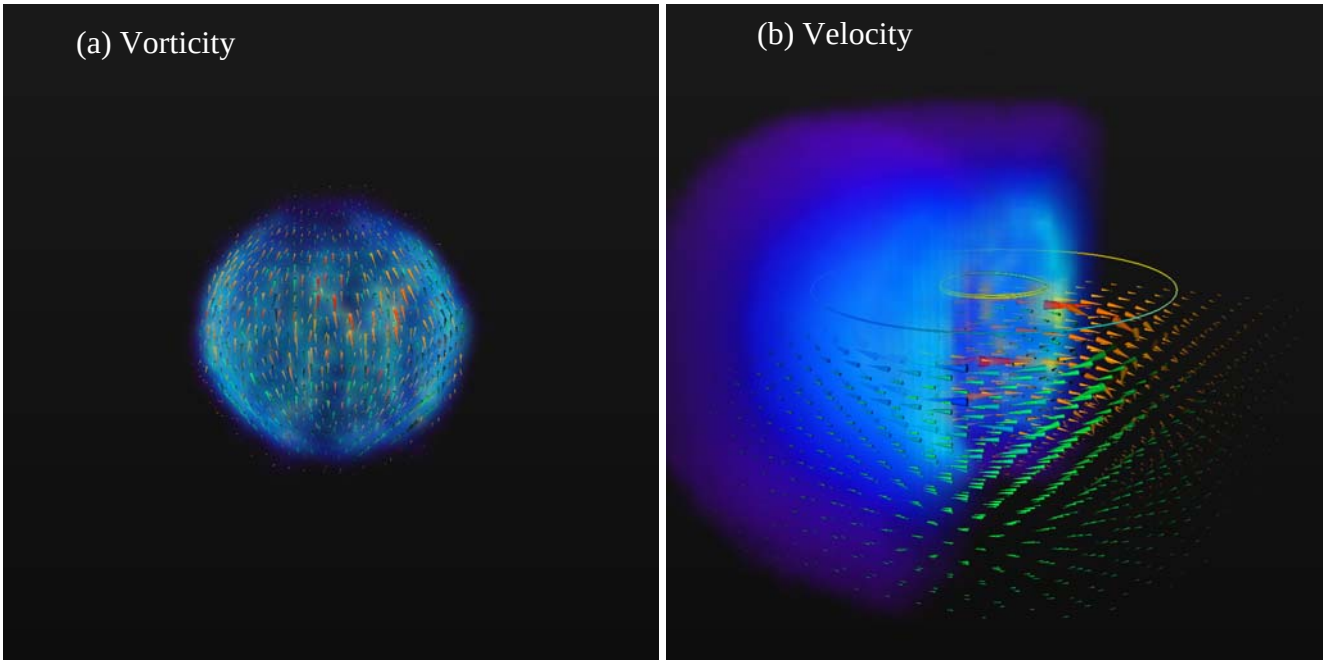


Figure 2. Ball spinning in fluid causes (a) vorticity and (b) a velocity field to form.

Figure 3 shows a few frames of animation of the cloud of passive tracer particles swirling around the spinning ball. Still frames really do not do justice to this so I strongly encourage you to download, build and run the code that accompanies this article.

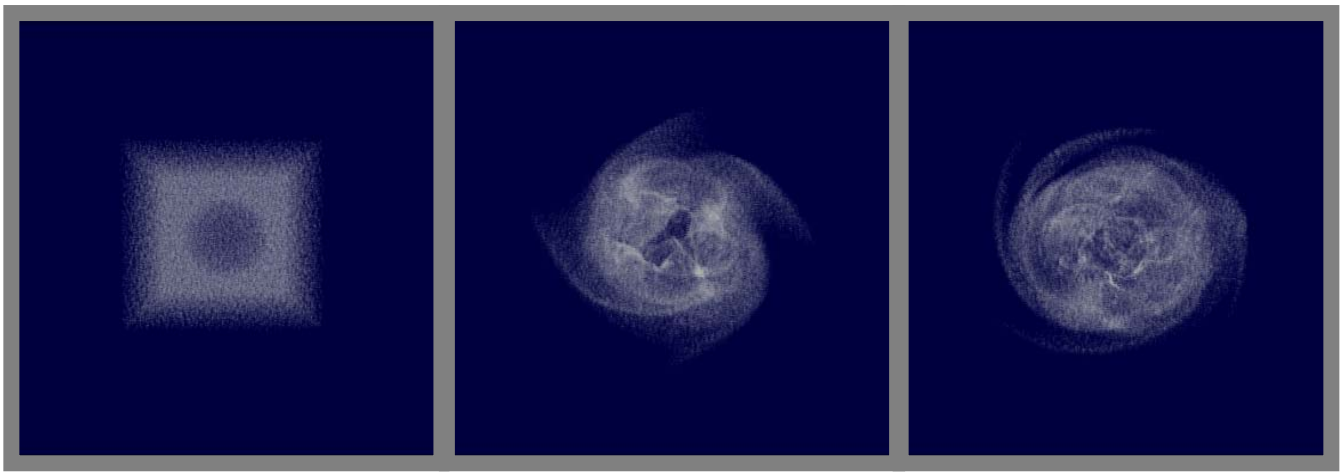


Figure 3. Top view of ball spinning in fluid. Still images do not do this justice.

The last scenario has a ball moving through fluid, generating a wake behind it, shown in Figure 4. Once again, the only thing causing any fluid to move is the thin layer of vorticity that occurs as a result of satisfying the boundary conditions on the surface of the ball. From this, vortices shed from the body, and a wake forms behind it. Visual comparisons with Gourlay, *et al.* (2001) show that this simulation produces the expected wake configuration behind the body. If you are so inclined, you could further validate the results of this simulation by comparing characteristics of the late wake, far behind the body, to see how they compare with theory and experiments in real fluids, which are documented, for example, in Johansson, *et al.* (2003).

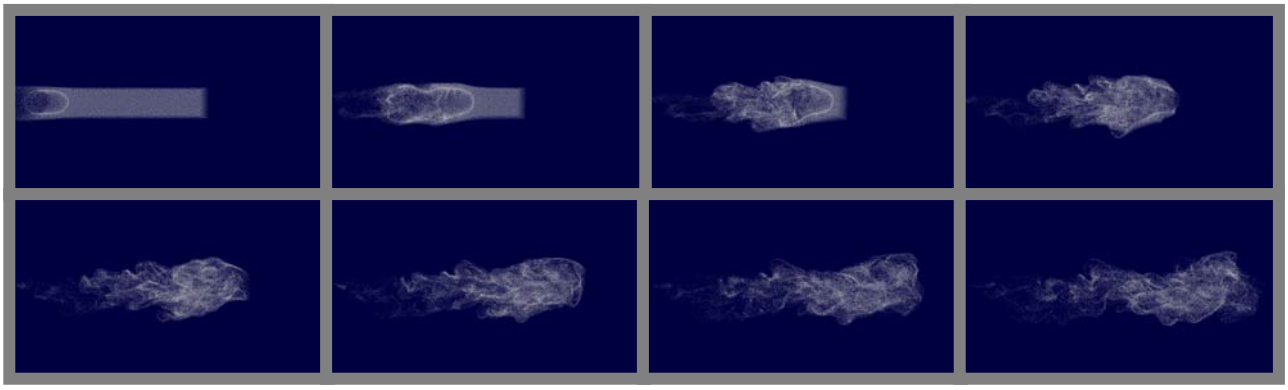


Figure 4. Invisible ball passes through a block of smoke particles. The simulation runs and renders faster than 60 frames per second. You must see this running live to appreciate it.

Figure 5 shows a plot of the motion of the ball as it passes through the fluid, showing that the fluid does indeed induce an aerodynamic drag force on the ball—that is, the ball slows down over time.

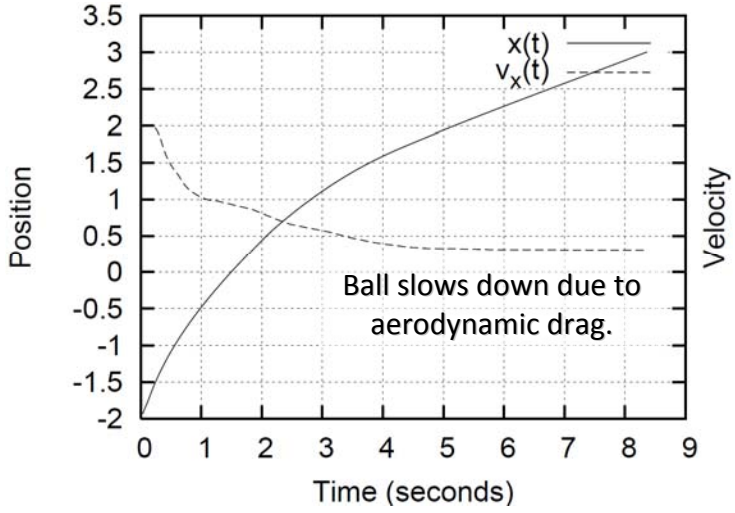


Figure 5. Motion of a ball as it passes through fluid

The next article in this series focuses on performance analysis and comparisons. Meanwhile, I simply report that these simulations run and render faster than 30 frames per second on a modest laptop using a single 2.5 GHz core of an Intel® Core™2 Duo processor and faster than 60 frames per second—with only 35% CPU utilization—on a 2.33 GHz dual Quad-Core Intel® Xeon® processor. This simulation runs fast, and after future articles its speed and utility will increase.

Summary

This article presents a novel formulation for dealing with fluid–body interactions, which entails assigning vortex elements to satisfy velocity boundary conditions. The simulation accumulates changes in angular and linear momentum applied to particles, and then applies the same changes, via impulses, to rigid bodies, providing a two-way coupling between the fluid and bodies immersed in it. Finally, this simulation exploits the embarrassingly data-parallel nature of the algorithms and uses Intel® TBB `parallel-for` construct to spread the computational cost across multiple threads.

The resulting simulation runs and renders all of its demonstration cases faster than 30 frames per second on a modest laptop and faster than 60 frames per second on a multi-core desktop computer.

As impressive as (I hope) these results might seem, they fall short of the requirements of a video game. Although these simulations run at interactive speeds, as presented, they leave

insufficient computational resources for the rest of the game. (The rendering methods presented here also leave something to be desired, although these articles focus on simulation, not rendering.) And furthermore, any in-game technology should have an accompanying content-creation tool and asset conditioning pipeline. Future articles will address these issues.

The next article in this series will analyze performance using Intel® VTune™ Performance Analyzer. Vector and matrix operations pervade this simulation code, so perhaps Intel® Streaming SIMD Extensions (SSE) could help enhance performance. Also, some of the routines that (in the code that accompanies this article) remain serial, can be written as multi-threaded using more sophisticated threading techniques.

Eventually, this series of articles will explore more aggressive optimization techniques that boldly depart even further from the realm of rigorous computational fluid dynamics and into more artistic endeavors, where the goal is to make high-performance, interesting motion—even if it means that the resulting simulation does not stand up to scientific scrutiny but does result in appealing motion and visual effects at an affordable cost.

Further Reading

Cottet, GH., and P.D. Koumoutsakos. 2000. *Vortex Methods: Theory and Practice.* Cambridge: Cambridge UP.

Gourlay, M.J., S.C. Arendt, D.C. Fritts, and J. Werne. 2001. Numerical modeling of initially turbulent wakes with net momentum. *Physics of Fluids* 13(12):3783–802.

Johansson, P.B V., George, W.K. & Gourlay, M.J. (2003): Equilibrium similarity, effects of initial conditions and local Reynolds number on the axisymmetric wake. Phys. Fluids 15, 603, 22 January.

Reinders, J. (2007): *Intel Threading Building Blocks*. O'Reilly Media, Sebastopol, CA.