Fluid Simulation for Video Games (part 12)

By Dr. Michael J. Gourlay

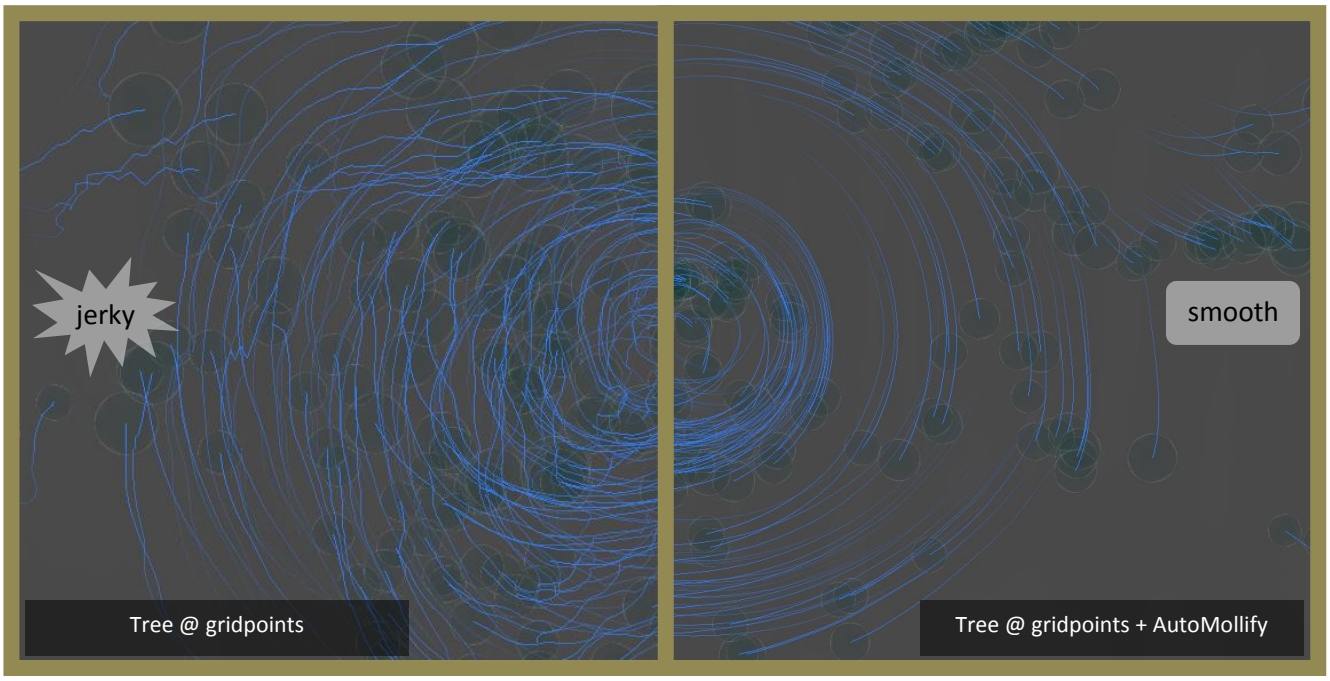

Figure 1. Comparison of particle paths for different velocity-from-vorticity methods. The left side shows jerky paths from before this article. The right side shows smooth paths after changes this article introduces.

Jerkstrophy

It comes down to sampling.

Throughout this article series, I have explained the many liberties taken with simulation algorithms for the sake of speed. This article explores one ugly result of those liberties: jerky motion.

This article—the twelfth in a series—explains how improper sampling causes unwanted jerky motion and describes how to mitigate it. 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 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 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.

Note: *Jerkstrophy* is an Engreek portmanteau of the English *jerk,* meaning sudden uncontrolled movement, and the Greek *strophy,* meaning rotation.

Determinism

Determinism means that given the same input, the simulation generates the same output. Furthermore, the results you obtain from parallel code should be identical to those results obtained with serial code. The code presented prior to this article did not have that property when it ran on multiple threads. This article explores why and explains how to correct that using a feature of Intel® Threading Building Blocks (Intel® TBB) called `parallel_invoke`.

Diagnosis

What causes jerking in the vorton fluid simulation that these articles describe?

Eliminate some possibilities: Jerking occurs even if you disable buoyancy, stretching, viscous diffusion, and thermal diffusion. That leaves advection, which depends on velocity. That conclusion suggests that the velocity-from-vorticity algorithm causes jerking.

The demo code includes multiple velocity-from-vorticity techniques and algorithms. The differential techniques that use the Poisson solver do not show jerking. That leaves the integral techniques.

Both the direct solver and the treecode show the jerky behavior. They share a few aspects: the Biot-Savart formula that gives velocity from vorticity and position and the fact that both techniques evaluate velocity at grid points, yet sample it at vortons (see Figure 2).

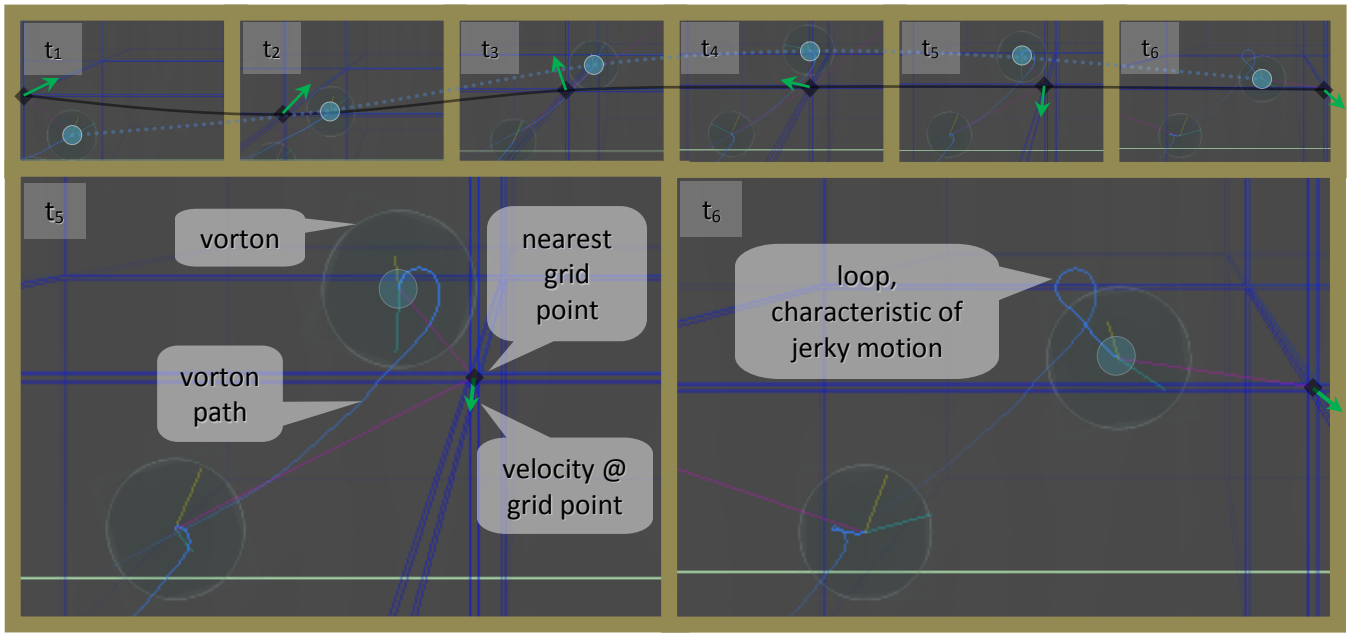

Figure 2. Vorton moving in a loop. Each vorton induces a velocity field, evaluated at grid points. The nearest grid point has the greatest influence on the vorton and vice versa. As the highlighted vorton and its nearest grid point move relative to each other, the vorton loops around. The dotted turquoise curve across the top row simply guides the eye to follow the highlighted vorton across adjacent frames. Likewise, the black curve across the top row guides the eye to see the grid point (black diamond) nearest the highlighted vorton. The top row shows that the vorton moves counterclockwise relative to its nearest grid point.

What cause this jerking? Figure 2 shows a zoomed-in example of a single "jerking event." It shows a vorton passing nearby a grid point and looping. The vorton induces a velocity field throughout space, but the simulation only evaluates that field at grid points. The algorithm, in turn, interpolates the velocity field using the eight grid points that surround the vorton to obtain a velocity value at the vorton. But this is inaccurate.

If the fluid consisted of a single vorton, that vorton would never move; the velocity field induced by a vorton, on itself, is zero. But in this simulation, it is non-zero.

What's worse, the grid points and vorton both move, but for different reasons. Vortons move as a result of advection; grid points move as a result of the domain migrating. Not only does each vorton incorrectly influence its own motion, but the self-motion is influenced by factors unrelated to the local vorton motion. As the vorton and grid point move relative to each other, the velocity changes *direction,* and that appears as jerking.

Possible Solutions

Jerking is caused by evaluating velocity at grid points, then interpolating at vortons. If the simulation evaluated velocity at vortons, the jerking should stop. And indeed it does, as the results below demonstrate.

But evaluating velocity at vortons comes with a hefty performance cost. The velocity field that a vorton induces is most intense at the periphery of the vorton. Figure 3 shows the velocity profile of a single vorton. Another way to consider this problem is that the simulation should resolve vortons and grid points as being approximately at the same place; any length scale smaller than a grid cell is under-resolved. And indeed, as the domain expands, the vorton radii eventually become much smaller than grid cells (or rather, grid cells expand to become much larger than vorton radii—the radii remain constant).

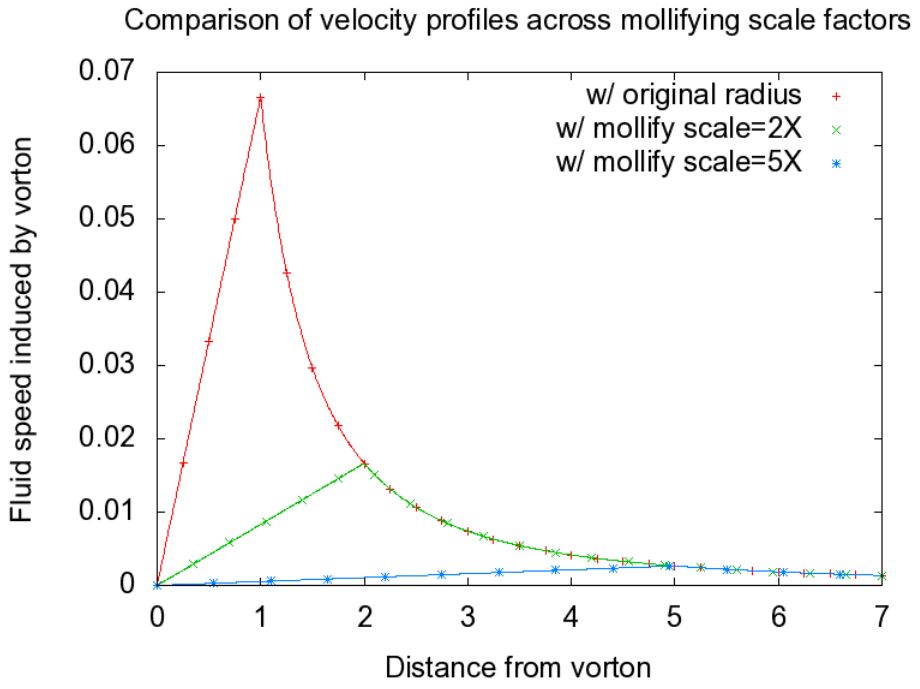

Figure 3. Velocity profiles for various mollifying scale factors

Perhaps you could spread vorton radii to be commensurate with grid cell sizes. (I call this *auto-mollification.*) Figure 3 also shows the velocity profiles for vortons whose radii have been spread by a factor of 2× and 5×. Note that outside the vorton radius, the velocity profile is identical for all three cases; but inside the vortex core, spread vortons yield smoother velocity profiles. That means that this change only affects things that are too small (or too close) to resolve; large-scale (or long-distance) influence remains the same as before.

Note: *Mollify* means "to soften," and in mathematics, a *mollifier* is a smooth function to convolve with a non-smooth function. I borrowed the term here, though my usage differs somewhat from canon.

Yet another possible solution to avoid jerking is to use the differential velocity solver. But it suffers from having an undue influence from boundary conditions. You could push out the boundaries to approximate a free fluid (one far from boundaries), but that would require increasing the number of grid points (increasing computational and memory costs), reducing resolution (resulting in less detail), or using an adaptive grid (requiring slow, complicated algorithms). Try it and see what happens; the results are below.

Smoothing Jerky Motion

There are many hypothetical solutions to reducing jerking. A scientist or engineer would tend toward solutions that lead to higher accuracy, even at the cost of speed. After all, in science, a faster result is useless if it is wrong. But whereas in science *wrong* means "inaccurate," in visual effects, *wrong* means "ugly." And *ugly* includes *slow* and *distracting.* So, of the possible solutions, game developers care more about speed and plausibility than physical accuracy.

Evaluate Velocity at Vortons Instead of at Grid Points

As a sanity check, change the direct solver algorithm to evaluate velocity at vortons (instead of at a grid point). Then, vortons will have zero self-influence (which is correct). See Figure 4.

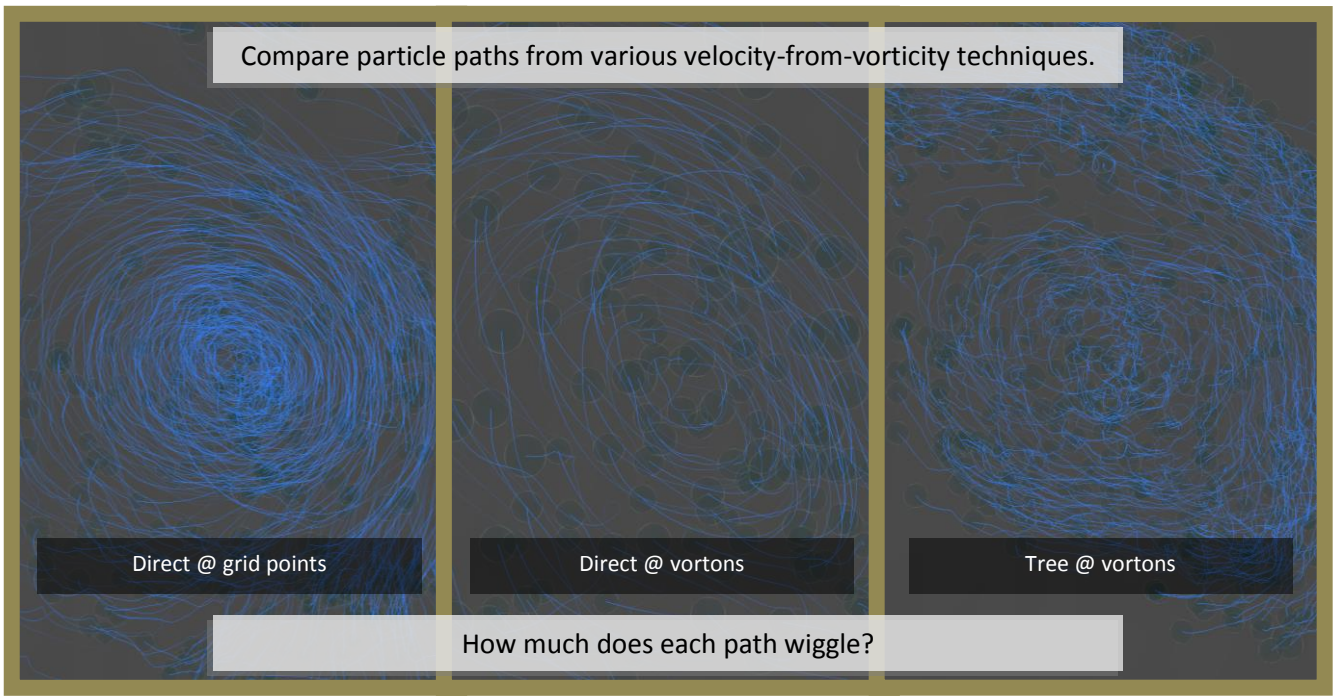

Figure4. Vorton paths evaluating velocity at grid points versus at vortons. Each panel shows a gyre from a different simulation.

Figure 4 shows the paths vortons take as they move throughout the fluid. *Direct @ grid points* uses direct summation, evaluating velocities at grid points, then interpolates velocity at vortons—the technique described in part 3 and that results in jerky motion. The path lines wiggle intensely, revealing jerky motion, and this is the problem we seek to eliminate.

Direct @ vortons evaluates velocity at vortons—the numerically correct approach—and the path lines are much smoother. This approach solves the jerking problem but at a great computational cost; this algorithm is $O(N^2)$—too high. But use *Direct @ vortons* as the basis of comparison for all other techniques.

Tree @ vortons uses the treecode (O(NlogN), tolerably fast) but evaluates velocity at vortons instead of at grid points. Surprise! The vortons still jerk. Why? Remember from part 3 that the treecode creates a tree of "supervortons" whose positions are weighted sums of actual vorton positions. Even at the leaf layer of the tree, the grid algorithm creates and uses supervortons. This means that a supervorton will often be just barely adjacent to the vorton sampling the velocity field. Effectively, vortons influence themselves again, so we're back to the original problem. You can solve that by using original vortons (instead of supervortons) at the leaf layer. The sample code accompanying this article supports that feature (if you enable `USE_ORIGINAL_VORTONS_IN_BASE_LAYER`), and it yields smoother motion. But it also slows the simulation.

The hypothesis is that improperly sampling velocity causes jerky motion. This experiment corroborated that hypothesis, but the solution costs too much. Try another approach.

Move Vortons to Grid Points Before Evaluating Velocity

Part 2 briefly mentioned a technique called *particle-in-cell* (PIC) but did not elaborate. PIC algorithms use particles to carry vorticity and the particles move around, but each frame their position is reset so that each grid cell has a single particle. Likewise, you could move vortons to grid points and evaluate velocity at grid points. The process consists of these steps:

1. Create a vorticity field by populating a grid based on vorticity from vortons.
2. Create a new vorton at each grid point using that vorticity. (Ignore the original vorton after that.)
3. Use an integral technique to compute velocity at each grid point (same as before).
4. (Optional) Dispose of the new vortons and reuse the original vortons.

Note: If, after computing the velocity field, you dispose of the regridded vortons and retain the original vortons, the algorithm is called "full particle" particle-in-cell. The FLuid-Implicit-Particle (FLIP) method is one variant of this.

Because the vortons and grid point would coincide, the new vortons have no self-influence, thus reducing the attendant near-singular velocity field. (See Figure 5.)

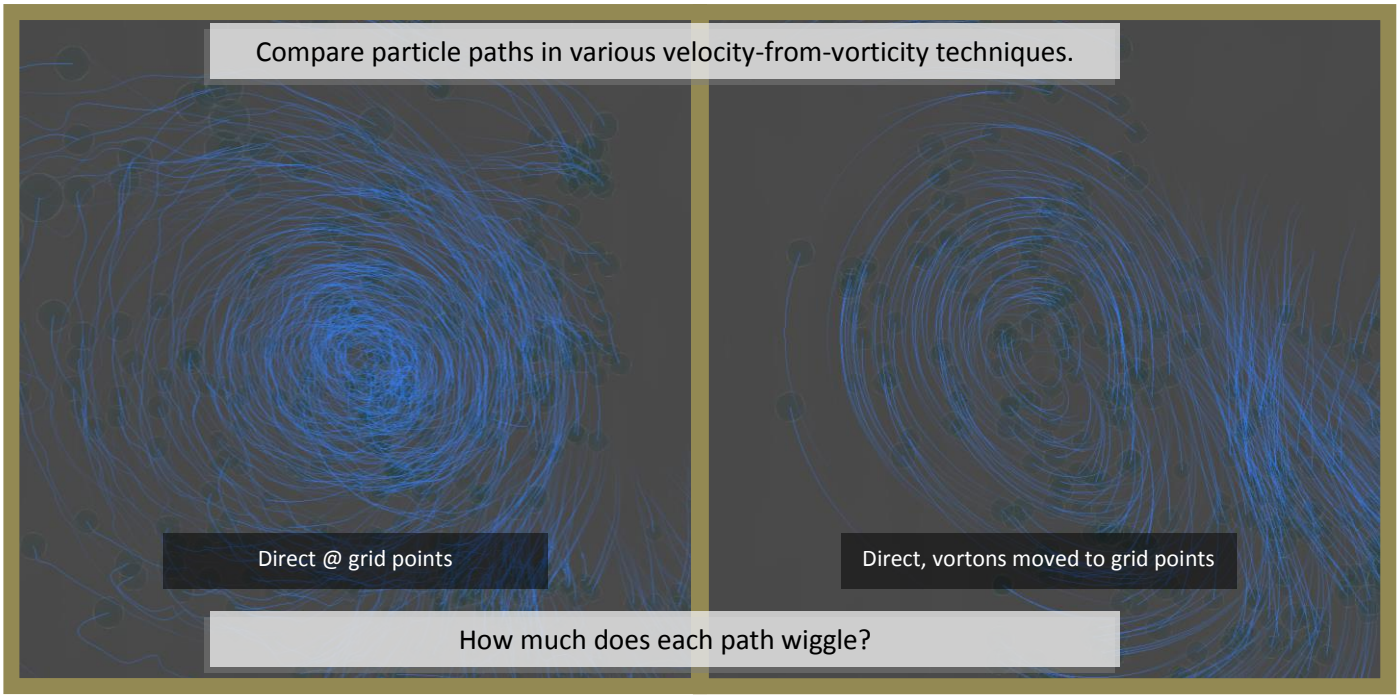

Figure 5. Vorton paths evaluating velocity at grid points after moving them to grid points

Figure 5 compares path lines for this technique. *Direct @ grid points* is as in Figure 4. *Direct, vorton moved to grid points* shows path lines from using this technique involving moving vortons to grid points. As expected, the motion is smoother.

This technique suffers from the same problems as the former: Direct summation is too slow, and trying to apply the same technique to treecode results in no improvement.

Try another approach: Smooth the vorticity field.

Smooth Vorticity Field: Auto-mollification

Evaluating velocity at grid points instead of at vortons causes jerking, but the algorithm does so for a good reason: It expedites sampling velocity anywhere in space. We really just want grid points and vortons to coincide. But remember, this is a discretized problem with finite resolution, trying to represent a continuous problem with infinite resolution. The simulation needs to respect that discretization; the velocity field should not have spikey features that the grid cannot resolve. The vorticity distribution should be smooth enough for the grid to resolve.

Spreading vorton radii to be commensurate with grid cell sizes is tantamount to representing vorticity to about the same length scale the grid can resolve. Without this, vortons represent

vorticity on too fine a scale, which leads to excessively rapid and highly detailed motion that appears as jerking (see Figure 6).

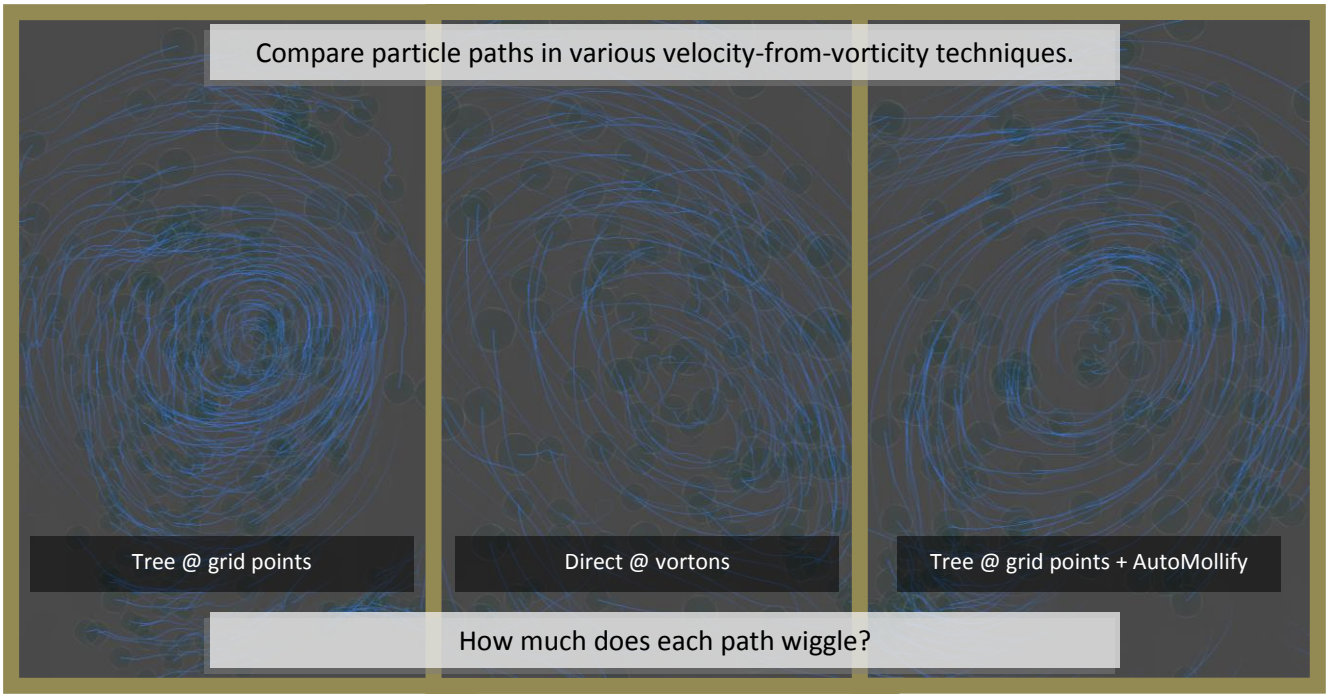

Figure 6. Vorton paths comparing original and mollified velocity profiles

Figure 6 shows path lines for the original fast but jerky *Tree @ grid points,* the accurate but slow *Direct @ vortons,* and the fast and smooth *Tree @ grid points + AutoMollify,* which spreads each vorton to a size that the grid can resolve. The result is motion without jerking at about the same speed as the original algorithm.

Success!

The following code snippet shows the velocity-from-vorticity formula, with the auto-mollification highlighted in purple bold.

```
    const Vec3          vNeighborToSelf = vPosQuery - mPosition ;
    const float         radius          = mSize * 0.5f * mSpreadingRangeFactor ;
    const float         radius2         = radius * radius ;
    const float         dist2           = vNeighborToSelf.Mag2() ;
    const float         distLaw         = ( dist2 < radius2 )
                                            ?   /* Inside vortex core */
                                            ( 1.0f / ( radius2 * radius ) )
                                            :   /* Outside vortex core */
                                            ( finvsqrtf( dist2 ) / dist2 ) ;
    const Vec3  velocityContribution =
        TwoThirds * radius2 * radius * mAngularVelocity ^ vNeighborToSelf * distLaw ;
    vVelocity +=  velocityContribution * mSpreadingCirculationFactor ;
```

To conserve total circulation, vorticity must scale as the inverse of the cube of vorton radius; the spreading circulation factor, *C,* must equal the spreading range factor, *R,* to the negative third power. That is because, in the formula above, *R* appears cubed in the numerator and *C* linearly in the denominator, and you want $R^3/C \equiv 1$ so $C = R^{-3}$.

Each frame, you should assign the spreading range factor such that each vorton occupies approximately the same volume as a grid cell.

You can fine-tune the auto-mollification scale factor, as described below. But first, revisit the differential solver.

Differential Velocity Solver with Expanded Boundaries

The differential velocity solver presented in part 6 is fast and smooth, but it has undue influence from boundary conditions. You can, however, expand domain boundaries to reduce their influence on the flow.

Without also increasing grid resolution, expanding the domain implies expanding each grid cell, which in turn implies that more vortons reside in each grid cell. The former particle strength exchange algorithm assumed that, on average, each grid cell would contain a single vorton (because the grid was constructed to satisfy that assumption). But expanding boundaries without increasing resolution would violate that assumption. So, the diffusion algorithm must fall off with distance; otherwise, vorticity and heat would be diffused far too rapidly. The code snippet below for `ExchangeVorticity` shows the new algorithm, where the bold purple text highlights the new falloff factor.

```
void VortonSim::ExchangeVorticity( Vorton & rVortonHere , Vec3 & rAngVelHere
    , const unsigned & ivThere , Vector< unsigned > & cell , const float & timeStep )
{
    const unsigned & rVortIdxThere   = cell[ ivThere ] ;
    Vorton &         rVortonThere    = (*mVortons)[ rVortIdxThere ] ;
    const float      diffusionRange  = 4.0f * rVortonHere.mSize ;
    const float      diffusionRange2 = POW2( diffusionRange ) ;
    const Vec3       separation      = rVortonHere.mPosition - rVortonThere.mPosition ;
    const float      dist2           = separation.Mag2() ;
    const float      distLaw         = exp( - dist2 / diffusionRange2 ) ;
    Vec3 &           rAngVelThere    = rVortonThere.mAngularVelocity ;
    const Vec3       vortDiff        = rAngVelHere - rAngVelThere ;
    const Vec3       exchange        = 2.0f * mViscosity * timeStep * vortDiff * distLaw ;
    rAngVelHere  -= exchange ;   // Make "here"  vorticity a little closer to "there".
    rAngVelThere += exchange ;   // Make "there" vorticity a little closer to "here".
}
```

Compare the differential solver with the integral solvers. See Figure 7.

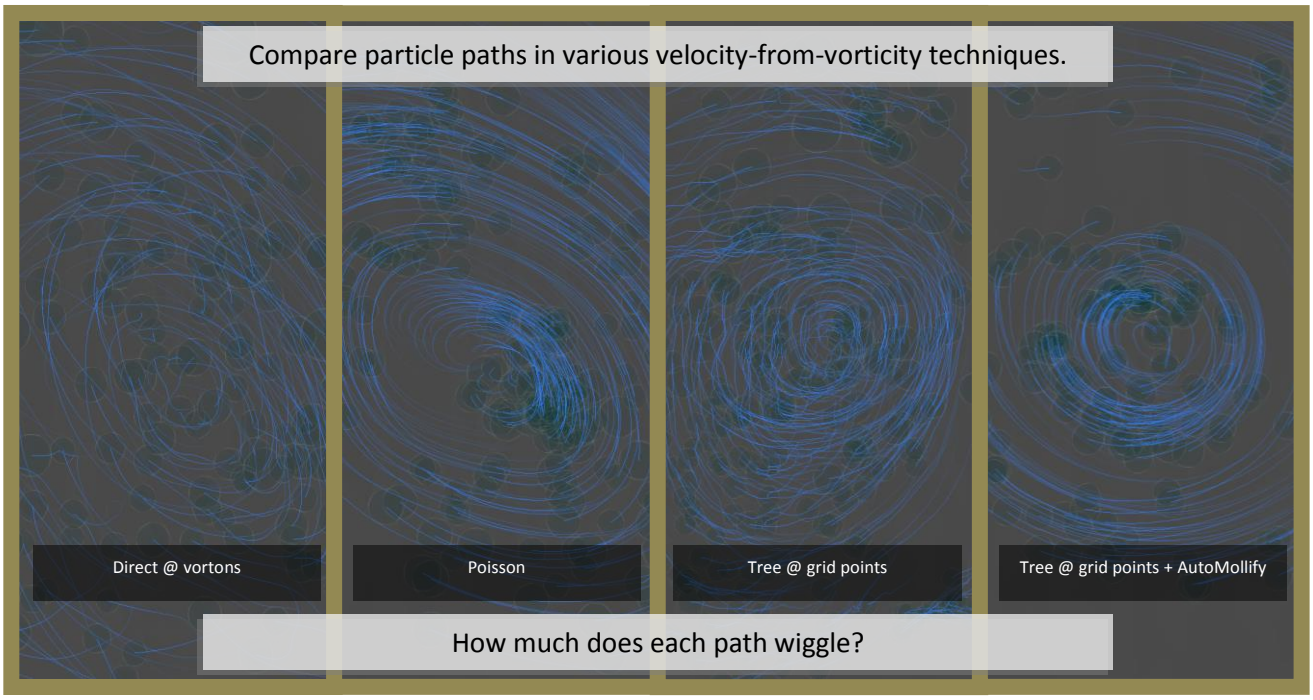

Figure 7. Vorton paths comparing integral and differential techniques

Figure 7 shows path lines for the various techniques this article explores. The differential technique, Poisson, shows smooth path lines, which is what you want.

Compare the differential solver with tight and expanded boundaries. See Figure 8.

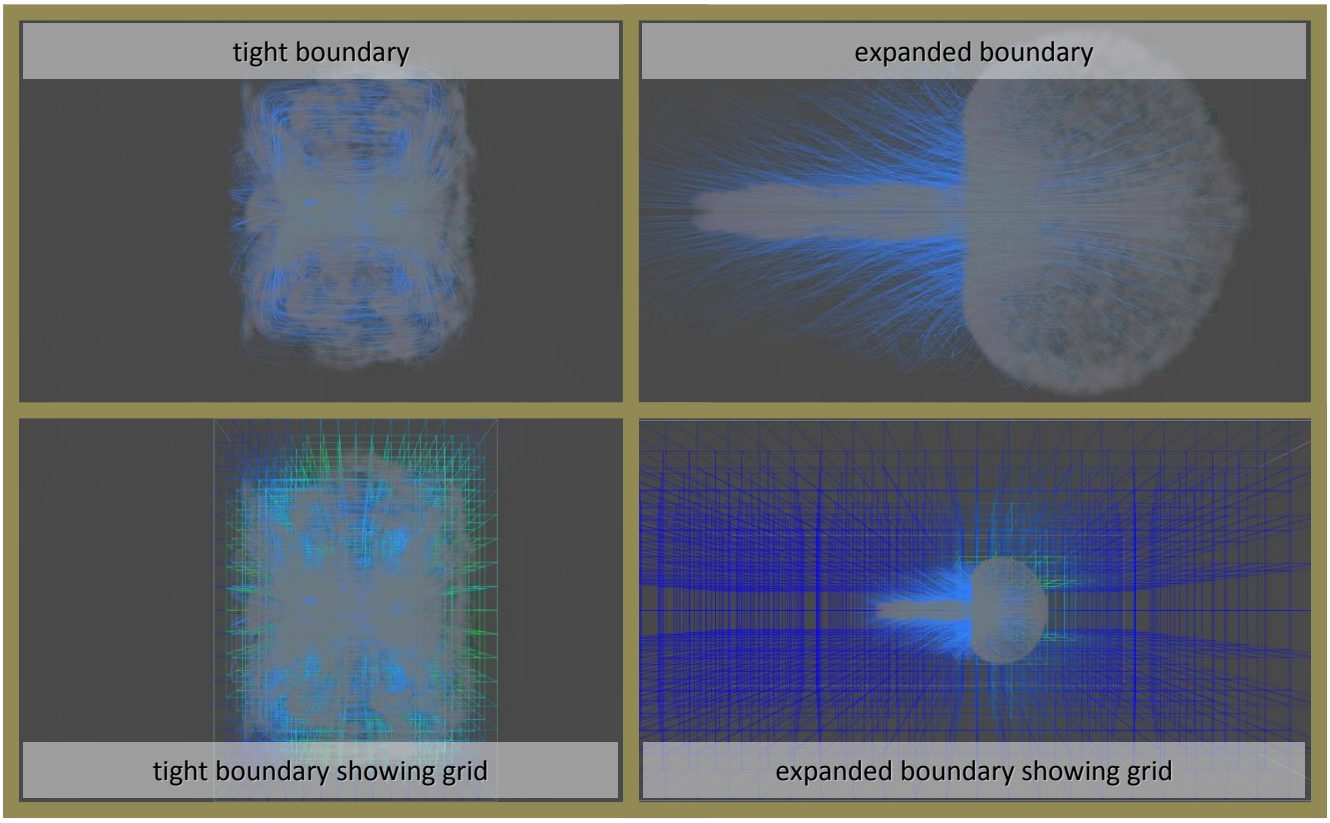

Figure 8. Path lines for a ring vorton simulated with a differential solver, with tight and expanded boundaries

Figure 8 shows a single-vortex ring propagating across the domain from left to right. With tight boundaries, the vortex ring barely moves and takes on a boxy shape; it is essentially a vortex ring inside a box. With expanded boundaries, the vortex ring propagates as it should. This works adequately for a simple flow like a single-vortex ring, but for more complicated flows, the larger and sparser grid cells leave the flow under-resolved and visually uninteresting.

Comparison of Techniques

The auto-mollification scheme has a tunable parameter, which is how large the vortons should be compared to a grid cell. The larger that parameter, the smoother the velocity field. What is the correct length scale? Whichever one gives results as close as possible to the direct summation approach.

Looking at path lines gives a qualitative sense of jerking, but a quantitative measure will remove some subjectivity.

How can you measure jerkiness? Physicists use the term *jerk* to mean the time derivative of acceleration. That gives a precise, quantitative measure of jerkiness, as shown in Figure 9.

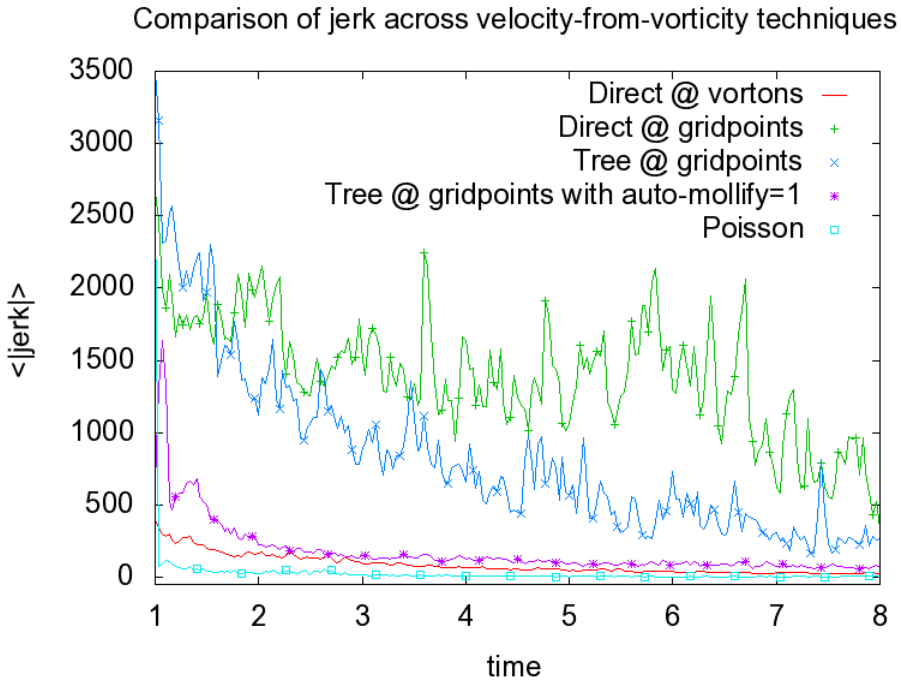

Figure 9. Comparison of jerk-over-time for various velocity-from-vorticity techniques

Figure 9 provides a comparison of the ensemble average of the magnitude of jerk of all vortons, over time, for various velocity-from-vorticity algorithms. Using the *Direct @ vortons* as a basis, the *Tree @ grid points with auto-mollify* looks promising, as does *Poisson*. That corroborates qualitative results from the path line plots.

Tuning Auto-mollification

Some jerkiness occurs naturally; completely jerk-free motion would look boring. The auto-mollification technique has a spreading-length scale you can use to tune how much jerkiness the simulation has.

The following code snippet shows how to compute the vorton radius spreading factors introduced earlier.

```
float gridLength = lengthScale * powf( mVelGrid.GetCellVolume() , 1.0f / 3.0f ) ;
mSpreadingRangeFactor       = MAX2( gridLength / mVortonRadius , 1.0f ) ;
mSpreadingCirculationFactor = 1.0f / POW3( mSpreadingRangeFactor ) ;
```

Consider what happens to jerk when `lengthScale` varies (see Figure 10).

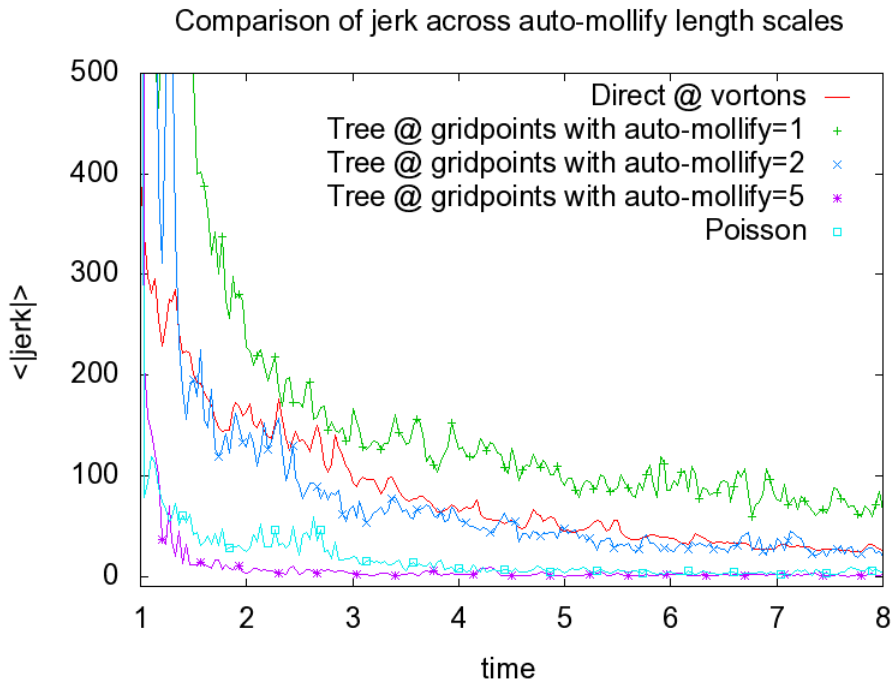

Figure 10. Comparison of jerk-over-time for each value of `lengthScale`

Figure 10 shows jerk over time for various values of the auto-mollification `lengthScale`. At `lengthScale=2`, the results approximately match those from the basis of comparison, *Direct @ vortons.* Larger values result in motion so smooth it looks boring. Smaller values result in jerky motion.

Diagnosing this problem required isolating specific scenarios and repeating them exactly. The simulation had to produce the same results each time. That turns out to be non-trivial.

Determinism

Determinism means that given the same inputs, the program generates the same outputs. When threaded, the fluid simulation algorithm presented in former articles is not deterministic, which makes diagnosing problems like jerking difficult, because you want to repeat an event like that in Figure 2 to examine it closely.

The non-linear nature of fluid advection exacerbates non-determinism, because even tiny initial differences rapidly lead to exponentially larger differences later. The demo program had multiple sources of non-determinism when run in parallel, all of which are fairly common when dealing with multithreaded algorithms.

Race Condition

The parallel particle strength exchange (PSE) algorithm presented in part 3 is not deterministic, because each particle is visited multiple times, potentially by different threads, and the threads are not spawned deterministically. In fact, the PSE algorithm presented previously is not even thread-safe, because it is conceivable that multiple threads could access the same particle simultaneously.

The parallel PSE algorithm divides work across slices distributed along the *z*-axis. Each iteration accesses particles in both the current and next *z*-slice, but another thread could "own" the next slice. That's the problem: Two threads can access the same data at around the same time, but the precise timing depends on preemptive scheduling, over which the program has no control. The results end up seeming random. The randomness manifests as tiny variations in particle parameters—perhaps one part in millions. But those tiny variations grow rapidly over time.

The easiest way to solve this problem would be to eliminate the access of neighboring *z*-slices. Doing so would eliminate non-determinism at the cost of preventing particles from exchanging vorticity very far along the *z*-axis. That would make diffusion anisotropic, which is not physically correct. Fortunately, solving this properly costs little CPU time and demonstrates a useful technique: odd–even communication.

The following code snippet shows how to modify the loop that iterates over the *z* direction of the grid—the direction across which the loop is parallelized. This same code occurs in both `DiffuseVorticityPSESlice` and `DiffuseHeatPSESlice`. The `phase` can be `PHASE_EVEN`, `PHASE_ODD` or `PHASE_BOTH`, where `PHASE_BOTH` yields the original serial algorithm. The bold purple code is new.

```
    const size_t zInc    = PHASE_BOTH == phase ? 1 : 2 ;
    const size_t flipper = ( PHASE_ODD == phase ) ? 1 : 0 ;
    const size_t izShift = ( izStart & 1 ) ^ flipper ;
    size_t idx[3] ;
    for( idx[2] = izStart + izShift ; idx[2] < izEnd ; idx[2] += zInc )
    {   // For all points along z except the last...
        ...
    }
```

Instead of computing diffusion in a single pass, use two passes—odd and even—like so:

```
parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize ) ,
 VortonSim_DiffuseHeatPSE_TBB( timeStep , this , ugVortonIndices , VortonSim::PHASE_EVEN ) );
parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize ) ,
 VortonSim_DiffuseHeatPSE_TBB( timeStep , this , ugVortonIndices , VortonSim::PHASE_ODD  ) );
```

These changes make the diffusion step deterministic when using multiple threads.

Floating-point Operations Are Not Associative

An *associative operation* is one in which order does not matter. Mathematically, associativity means:

$$a + b + c = a + (b + c) = (a + b) + c$$

That is, the order in which you perform the operation does not change the results. For integers and real numbers, addition and multiplication are associative.

Unfortunately, the computer implementation of floating-point operations such as addition and multiplication are not associative. Because floating-point numbers have finite precision, each operation truncates some value, so order matters. For example, if $a \gg b \gg c$, it is possible that $a + b$ is so large that the $+c$ term has no influence on $a + b$, but if you first add $b + c$, part of c's influence might carry over into the $+a$ term. You could choose $a = 1$, $b = \epsilon$, $c = 9\epsilon/10$ to see this, where ϵ is defined as the smallest floating-point number you can add to 1, such that $1 + \epsilon \neq 1$.

Recall that part 4 introduced an algorithm that computes two-way interactions between the fluid and bodies immersed in it. That algorithm used a reduction operation and Intel® TBB `parallel_reduce` to parallelize it. It turns out that `parallel_reduce` does not split and join tasks deterministically.

To make fluid–body interaction deterministic, use `parallel_invoke` for `FluidBodySim_CollideTracers_TBB`:

```
Vec3 FluidBodySim::CollideTracersReduce( Vector< Particle > & particles
  , float ambientFluidDensity , const RbSphere & rSphere , const size_t iPclStart
  , const size_t iPclEnd , const size_t grainSize )
{
    Vec3 impulseOnBody ;
    const size_t span = iPclEnd - iPclStart ;
    if( span <= grainSize )
    {   // Sub-problem fits into a single serial chunk.
        CollideTracersSlice( particles , ambientFluidDensity , rSphere , impulseOnBody
                           , iPclStart , iPclEnd ) ;
    }
    else
    {   // Problem is large enough to split into pieces.
        size_t iPclMiddle = iPclStart + span / 2 ;
        FluidBodySim_CollideTracers_TBB ct1( particles , ambientFluidDensity , rSphere
                                           , iPclStart  , iPclMiddle ) ;
        FluidBodySim_CollideTracers_TBB ct2( particles , ambientFluidDensity , rSphere
                                           , iPclMiddle , iPclEnd    ) ;
        tbb::parallel_invoke( ct1 , ct2 ) ;
        impulseOnBody = ct1.mImpulseOnBody + ct2.mImpulseOnBody ;
    }
    return impulseOnBody ;
}
```

The co-recursive routine `FluidBodySim::CollideTracersReduce` replaces the old `parallel_reduce` call from part 4. This version cooperates with the functor

(`FluidBodySim_CollideTracers_TBB`) to create a binary tree of tasks, where the leaf nodes perform the actual computation, and after each pair of threads finishes, the parent thread combines the results by summing them into `impulseOnBody`. The results are deterministic even when computed in parallel.

Note that the functor class `FluidBodySim_CollideTracers_TBB` constructor now takes two additional parameters: the loop begin and end indices. (For details, see the accompanying code.) The arguments to `parallel_invoke` are symbols that can be invoked like a function (including functions, function pointers, functors, or lambdas), but they must take no arguments (and ideally should return no value). To pass arguments through `parallel_invoke`, you can pass them to the constructor of the functor, as in the code snippet above.

Floating-point Control Word Propagation

Part 5 explained how to change the floating-point control word to gain performance optimizations. But that exposed a problem in the threading library: Worker threads do not necessarily adopt the floating-point control word (FPCW) and multimedia extension control status register (MMXCSR) from the master thread that spawns them.

When using Intel® TBB, you can fix this by having each Intel® TBB functor constructor store the FPCW and MMXCSR and each invocation operator set them. This helper routine gets the floating-point control word:

```
unsigned short GetFloatingPointControlWord()
{
  unsigned short ctrlWord ;
  __asm { fnstcw ctrlWord }
  return ctrlWord ;
}
```

This gets the multimedia extension control status register:

```
unsigned GetMmxControlStatusRegister()
{
  unsigned mxcsrVal ;
  _asm { STMXCSR mxcsrVal }
  return mxcsrVal ;
}
```

This sets the floating-point control word:

```
void SetFloatingPointControlWord( WORD NewCtrlWord )
{
  __asm { fldcw  NewCtrlWord }
}
```

This sets the multimedia extension control status register:

```
void SetMmxControlStatusRegister( unsigned newMxcsrVal )
{
  _asm { LDMXCSR newMxcsrVal }
}
```

This recipe shows how to write an Intel® TBB functor that propagates the FPCW and MMXCSR to each worker thread:

```
class ComputeSomething_TBB
{
    public:
        void operator() ( const tbb::blocked_range<size_t> & r ) const
        {   // Set FPCW and MMXCSR for each worker thread before doing any work.
            SetFloatingPointControlWord( mMasterThreadFloatingPointControlWord ) ;
            SetMmxControlStatusRegister( mMasterThreadMmxControlStatusRegister ) ;
            ComputeSomething_Slice( r.begin() , r.end() ) ;
        }
        ComputeSomething_TBB( ... )
        {   // Store the FPCW and MMXCSR from the master thread.
            mMasterThreadFloatingPointControlWord = GetFloatingPointControlWord() ;
            mMasterThreadMmxControlStatusRegister = GetMmxControlStatusRegister() ;
        }
    private:
        WORD        mMasterThreadFloatingPointControlWord   ;
        unsigned    mMasterThreadMmxControlStatusRegister   ;
} ;
```

After version 3, Intel® TBB automatically propagates the FPCW to worker threads, so perhaps in those versions, this extra code is not required. (See Floating-point Settings in Worker Threads May Differ from Master Thread for OpenMP, TBB and Intel Cilk Plus on the Intel® Software Network for more information.)

Summary

This article explained how to diagnose and mitigate jerking while maintaining speed. It also showed how to use an Intel® TBB feature called `parallel_invoke` to make a parallel algorithm deterministic.

Future Articles

Liquids take the shape of their containers on all but one surface, so modeling liquids also implies modeling containers. Future articles will include extending boundary conditions to include planes, convex hulls, and interiors, which will allow for creating containers. That will pave the way for a discussion of free surface tracking and surface tension—properties of liquids.

About the Author

Dr. Michael J. Gourlay works as a Senior Software Engineer at Electronic Arts. He currently works on character physics. He previously worked as the Software Architect for the Football Sports Business Unit, as a senior lead engineer on Madden NFL, on the procedural animation system used by EA, on Mixed Martial Arts (MMA), 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 (UCF) Florida Interactive Entertainment Academy (FIEA), 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 (CFD) 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 at Boulder.