Fluid Simulation for Video Games (part 20)

By Dr. Michael J. Gourlay

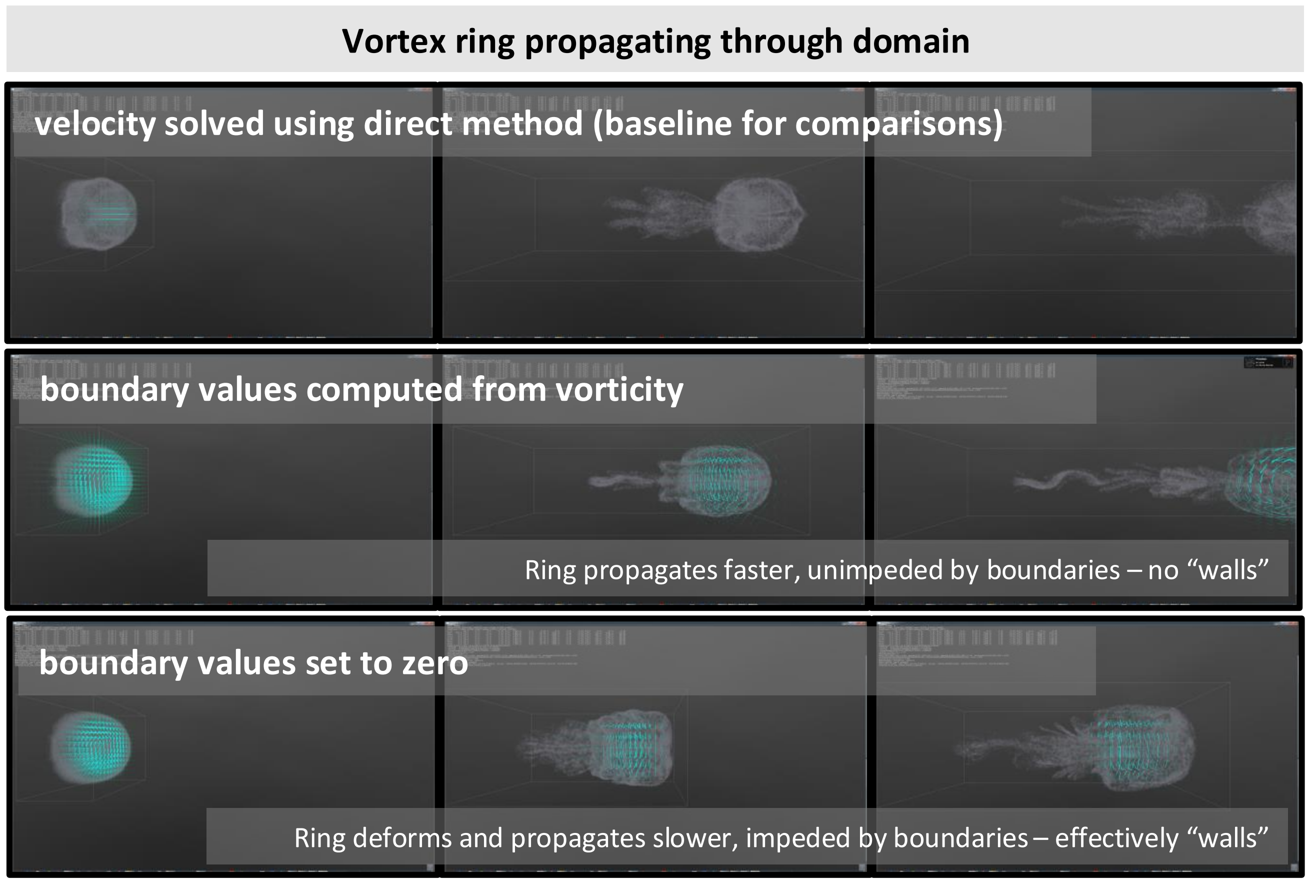
Vortex ring propagating through domain
velocity solved using direct method (baseline for comparisons)
boundary values computed from vorticity
Ring propagates faster, unimpeded by boundaries – no "walls"
boundary values set to zero
Ring deforms and propagates slower, impeded by boundaries – effectively "walls"

Figure 1: *Comparison of fluid simulations with the same interior vorticity but different solvers and boundary values. Top row uses direct integral method, where the domain boundary has no explicit influence. Middle row uses Poisson solver with boundary conditions of vector potential calculated from a treecode integral method. Bottom row uses Poisson solver with boundary conditions of vector potential set to zero.*

Assigning Vector Potential at Boundaries

Fluid simulation entails computing flow velocity everywhere in the fluid domain. This article places the capstone on a collection of techniques that combine to provide a fast way to compute velocity from vorticity. This is the recipe:

1. Within some domain, use vortex particles to track where fluid rotates.
2. At domain boundaries, integrate vorticity to compute the vector potential.
3. Throughout the domain interior, solve a vector Poisson equation to compute vector potential everywhere else.
4. Everywhere in the domain, compute the curl of vector potential to obtain the velocity field.
5. Advect particles according the velocity field.

This system has some nice properties:

- Most game engines already support particle systems. This technique builds upon such systems.
- Integrating vorticity can be accelerated using a treecode, which is O(N log N), which is pretty fast.
- Solving a vector Poisson equation can be even faster: O(N).
- Computing curl is mathematically simple and fast: O(N).
- Most particle systems already support advecting particles according to a velocity field.
- The same velocity field can advect both vortex and tracer particles.
- All the computation routines have fairly simple mathematical formulae.
- The algorithms are numerically stable.
- All the computation routines above can be parallelized using Intel Threading Building Blocks.
- Using particles to represent vorticity means only the most interesting aspects of the flow cost resources.

The scheme is well-suited to fire and smoke simulations but it has at least one a drawback: it not well-suited to simulating liquid-air interfaces. If you want to simulate pouring, splashes or waves, other techniques are better. Smoothed-particle hydrodynamics (SPH) and shallow-water wave equations will give better results.

This article complements the previous one to describe how to improve fidelity while reducing computational cost, by combining techniques described in earlier articles in this series. In particular, this article describes the following steps:

1. Compute vector potential at boundaries only, by integrating vorticity using a treecode.
2. Enable Multi-Grid Poisson solver described in Part 6.
3. Modify UpSample to skip overwriting values at boundaries.
4. Use TBB to parallelize UpSample and DownSample.

The code accompanying this article provides a complete fluid simulation using the vortex particle method. You can switch between various techniques, including integral and differential techniques, to compare their performance and visual aesthetics. At the end of the article I show some performance profiles that demonstrate that the method described in this article runs fastest of all those presented in this series. To my eye, it also gives the most visually pleasing motion.

Parts 1 and 2 summarized fluid dynamics and simulation techniques. Parts 3 and 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. Figure 1 shows the relationships between the various techniques and articles. Part 7 showed how to integrate a fluid simulation into a typical particle system. Parts 8, 9, 10 and 11 explained how to simulate density, buoyancy, heat and combustion in a vortex-based fluid simulation. 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. Parts 14, 15, 16, 17 and 18 added containers, smoothed particle hydrodynamics (SPH), liquids and fluid surfaces.

Integrate Vorticity to Compute Vector Potential at Boundaries

Part 19 explains details on how to use a treecode algorithm to integrate vorticity to compute vector potential. Use the treecode algorithm to compute vector potential only at boundaries, as shown in Figure 2. (Later, the vector Poisson solver will “fill in” vector potentials through the domain interior.)

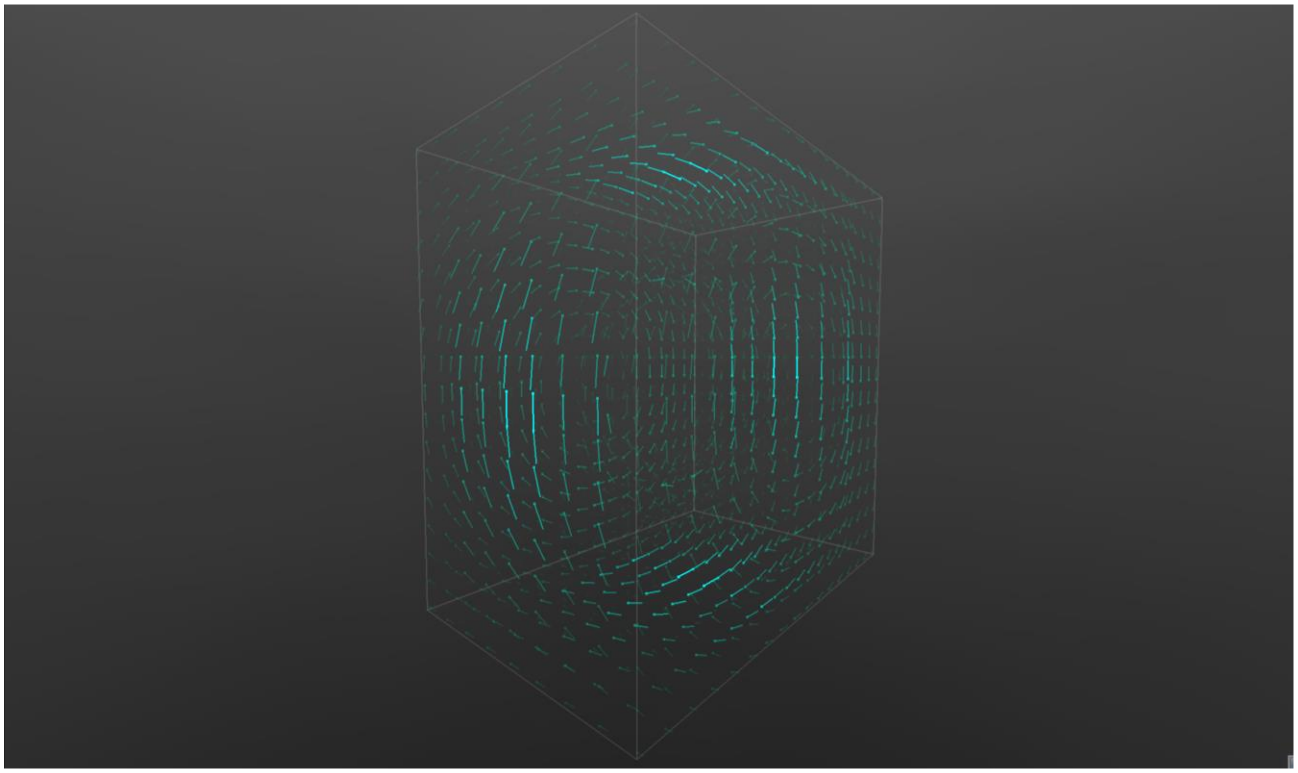

Figure 2: *Vector potential computed at domain boundaries only, for a vortex ring.*

The treecode algorithm has an asymptotic time complexity of O(N log N) where N is the number of points where the computation occurs. At first glance, that *seems* more expensive than the O(N) Poisson solver. But you can confine the treecode computation to the boundaries (a 2D manifold), which have N_b points, so the treecode algorithm costs $O(N_b \log N_b)$. In contrast, the Poisson algorithm runs in the 3D interior which has N_i points. (Figure 3 shows how the ratio of the numbers of boundary-to-interior points diminishes as the problem grows.) For a domain with $N_i \propto N_p^3$ points, $N_b \propto N_p^2 = N_i^{2/3}$, so the overall cost of this algorithm is

$$O(N_i^{2/3} \log N_i^{2/3}) + O(N_i)$$

The first term grows slower than the second, so asymptotically, the algorithm overall has asymptotic time complexity $O(N_i)$.

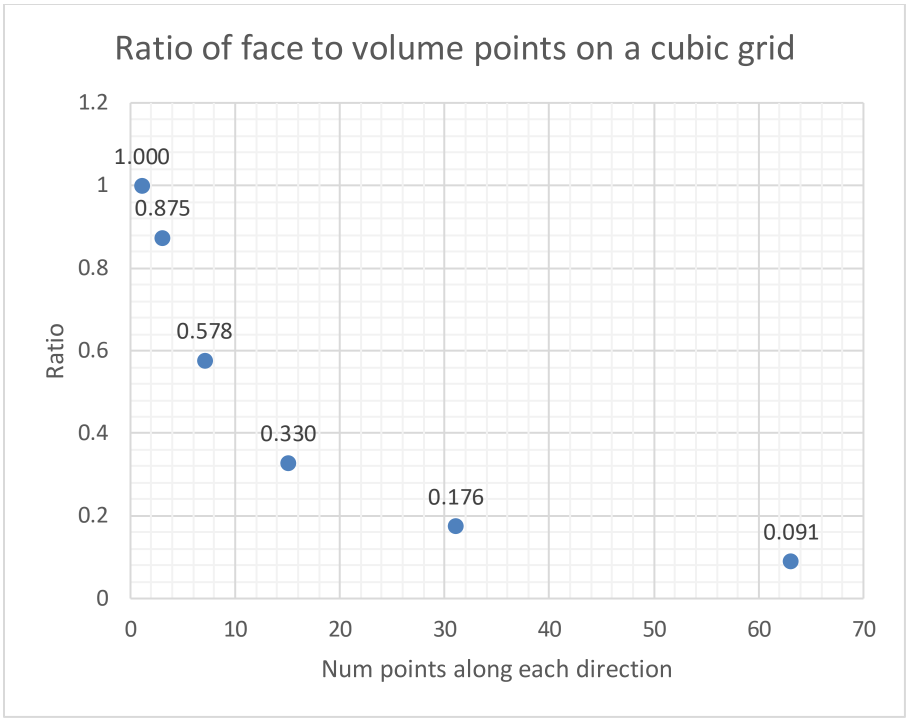

Figure 3: *Ratio of face to volume points on a cubic grid. As the number of grid points increases, the relative cost of computing boundary values diminishes, compared to computing interior values.*

Retain two code paths to compute the integral either throughout the entire domain, or just at boundaries. You can accommodate that by adding logic to conditionally skip the domain interior gridpoints. The yellow-highlighted code below shows that modification.

The treecode algorithm takes many conditional branches and its memory access pattern has poor spatial locality – it jumps around a lot. That makes the algorithm run slowly. Fortunately, vector potential values at boundaries have some properties you can exploit to reduce the cost of computing them: They don't vary much spatially, and they're far from most of the "action" in the domain interior. You can compute boundary values with lower spatial granularity to save some compute time. Achieve this by computing vector potential on boundaries at every *other* point, then copying those values to their neighbors. That cuts the cost about in half. The cyan-highlighted code below shows that modification.

(If you're curious about how much this decimated computation affects the final result, try computing with and without decimation. See if you can tell the difference.)

```
void VortonSim::ComputeVectorPotentialAtGridpoints_Slice( size_t izStart , size_t izEnd , bool boundariesOnly
                     , const UniformGrid< VECTOR< unsigned > > & vortonIndicesGrid , const NestedGrid< Vorton > & influenceTree )
{
    const size_t            numLayers               = influenceTree.GetDepth() ;
    UniformGrid< Vec3 > &   vectorPotentialGrid     = mVectorPotentialMultiGrid[ 0 ] ;
    const Vec3 &            vMinCorner              = mVelGrid.GetMinCorner() ;
    static const float      nudge                   = 1.0f - 2.0f * FLT_EPSILON ;
    const Vec3              vSpacing                = mVelGrid.GetCellSpacing() * nudge ;
    const unsigned          dims[3]                 =   { mVelGrid.GetNumPoints( 0 )
                                                        , mVelGrid.GetNumPoints( 1 )
                                                        , mVelGrid.GetNumPoints( 2 ) } ;
    const unsigned          numXY                   = dims[0] * dims[1] ;
    unsigned                idx[ 3 ] ;
    const unsigned          incrementXForInterior   = boundariesOnly ? ( dims[0] - 1 ) : 1 ;

    // Compute fluid flow vector potential at each boundary gridpoint, due to all vortons.
    for( idx[2] = static_cast< unsigned >( izStart ) ; idx[2] < izEnd ; ++ idx[2] )
    {   // For subset of z index values...
        Vec3 vPosition ;
        vPosition.z = vMinCorner.z + float( idx[2] ) * vSpacing.z ;
        const unsigned  offsetZ     = idx[2] * numXY ;
        const bool      topOrBottom = ( 0 == idx[2] ) || ( dims[2]-1 == idx[2] ) ;
        for( idx[1] = 0 ; idx[1] < dims[1] ; ++ idx[1] )
        {   // For every gridpoint along the y-axis...
            vPosition.y = vMinCorner.y + float( idx[1] ) * vSpacing.y ;
            const unsigned  offsetYZ    = idx[1] * dims[0] + offsetZ ;
            const bool      frontOrBack = ( 0 == idx[1] ) || ( dims[1]-1 == idx[1] ) ;
            const unsigned  incX        = ( topOrBottom || frontOrBack ) ? 1 : incrementXForInterior ;
            for( idx[0] = 0 ; idx[0] < dims[0] ; idx[0] += incX )
            {   // For every gridpoint along the x-axis...
                vPosition.x = vMinCorner.x + float( idx[0] ) * vSpacing.x ;
                const unsigned offsetXYZ = idx[0] + offsetYZ ;

                if( 0 == ( idx[0] & 1 ) )
                {   // Even x indices.  Compute value.
                    static const unsigned zeros[3] = { 0 , 0 , 0 } ; /* Starter indices for recursive algorithm */
                    if( numLayers > 1 )
                    {
                        vectorPotentialGrid[ offsetXYZ ] = ComputeVectorPotential_Tree( vPosition , zeros , numLayers - 1
                                                                , vortonIndicesGrid , influenceTree ) ;
                    }
                    else
                    {
                        vectorPotentialGrid[ offsetXYZ ] = ComputeVectorPotential_Direct( vPosition ) ;
                    }
                }
                else
                {   // Odd x indices. Copy value from preceding grid point.
                    vectorPotentialGrid[ offsetXYZ ] = vectorPotentialGrid[ offsetXYZ - 1 ] ;
                }
            }
        }
    }
}
```

Retaining boundary values when up-sampling

Interleaved between each solver step, multi-grid algorithms down-sample values from finer to coarser grids, then up-sample values from coarser to finer grids (as show in in Figure 3 and explained in Part 6). This resampling creates a problem: lower fidelity information up-sampled from coarser grids replace boundary values originally computed on finer grids.

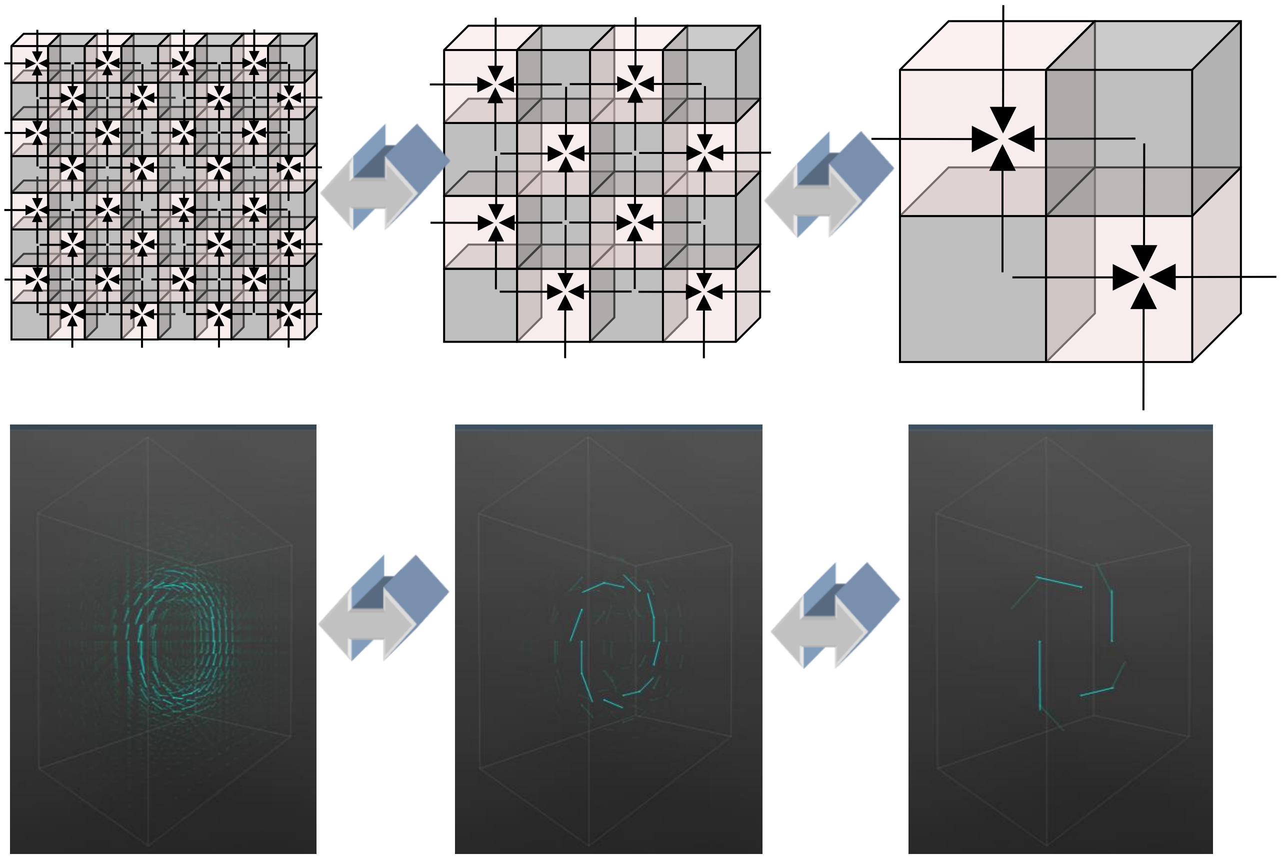

Figure 3. *A fine grid, a medium grid and a coarse grid in a multigrid solver.*

Since using the treecode to compute vector potential values is expensive, you want to avoid recomputing that. So, modify `UniformGrid::UpSample` to avoid overwriting values at boundaries. Use a flag in that routine to indicate whether to omit or include boundary points in the destination grid. To omit writing at boundaries, change the for-loop begin and end values to cover only the interior. (See the code for that below.) Then, in the multi-grid algorithm, during the up-sampling phase, pass the flag to omit up-sampling at boundaries. This code snippet is a modification of the version of `VortonSim::ComputeVectorPotential` originally presented in Part 6. The highlighted text shows the modification:

```
// Coarse-to-fine stage of V-cycle: Upsample from coarse to fine, running iterations of Poisson solver for each upsampled grid.
for( unsigned iLayer = maxValidDepth ; iLayer >= 1 ; -- iLayer )
{
    // Retain boundary values as they were computed initially (above) in finer grids.
    vectorPotentialMultiGrid.UpSampleFrom( iLayer , UniformGridGeometry::INTERIOR_ONLY ) ;
    SolveVectorPoisson( vectorPotentialMultiGrid[ iLayer - 1 ] , negativeVorticityMultiGrid[ iLayer - 1 ]
                      , numSolverSteps , boundaryCondition , mPoissonResidualStats ) ;
}
```

Avoiding superfluous and expensive intermediate fidelity when down-sampling

The downsampling routine provided in Part 6 accumulates values from multiple gridpoints in the finer source grid to compute values in the coarser destination grid. That provides higher fidelity results but since the Poisson solver overwrites those values with refinements, the additional fidelity is somewhat superfluous. It’s computationally cheaper to downsample using nearest values (instead of accumulating), then running more iterations of the Poisson solver (if you want the additional fidelity in the solution). So you can also modify DownSample to use a faster but less accuracy downsampling technique. This code snippet is a modification of the version of `VortonSim::ComputeVectorPotential` originally presented in Part 6. The highlighted text shows the modification:

```
  // Fine-to-coarse stage of V-cycle: downsample from fine to coarse, running some iterations of the Poisson solver for each
downsampled grid.
  for( unsigned iLayer = 1 ; iLayer < negativeVorticityMultiGrid.GetDepth() ; ++ iLayer )
  {
      const unsigned minDim = MIN3( negativeVorticityMultiGrid[ iLayer ].GetNumPoints( 0 )
          , negativeVorticityMultiGrid[ iLayer ].GetNumPoints( 1 ) , negativeVorticityMultiGrid[ iLayer ].GetNumPoints( 2 ) ) ;
      if( minDim > 2 )
      {
          negativeVorticityMultiGrid.DownSampleInto( iLayer , UniformGridGeometry::FASTER_LESS_ACCURATE ) ;
          vectorPotentialMultiGrid.DownSampleInto( iLayer , UniformGridGeometry::FASTER_LESS_ACCURATE ) ;
          SolveVectorPoisson( vectorPotentialMultiGrid[ iLayer ] , negativeVorticityMultiGrid[ iLayer ] , numSolverSteps
                            , boundaryCondition , mPoissonResidualStats ) ;
      }
      else
      {
          maxValidDepth = iLayer - 1 ;
          break ;
      }
  }
```

Parallelize Resampling Algorithms with Intel® Threading Building Blocks

Even with the above changes, the resampling routines cost a significant amount of time. You can use Intel® Threading Building Blocks to parallelize the resampling algorithms. The approach follows the familiar recipe:

- Write a worker routine that operates on a slice of the problem.
- Write a functor class that wraps the worker routine.
- Write a wrapper routine that directs TBB to call a functor.

The worker, functor and wrapper routines for DownSample and UpSample are sufficiently similar that I only include DownSample in this article text. You can see the entire code in the archive that accompanies this article.

This excerpt from the worker routine for DownSample shows the slicing logic, and modifications made to implement nearest-sampling, described above:

```
void DownSampleSlice( const UniformGrid< ItemT > & hiRes , AccuracyVersusSpeedE accuracyVsSpeed , size_t izStart , size_t izEnd )
        {
            UniformGrid< ItemT > &  loRes        = * this ;
            const unsigned  &       numXhiRes          = hiRes.GetNumPoints( 0 ) ;
            const unsigned          numXYhiRes         = numXhiRes * hiRes.GetNumPoints( 1 ) ;
            static const float      fMultiplierTable[]  = { 8.0f , 4.0f , 2.0f , 1.0f } ;

            // number of cells in each grid cluster
            const unsigned pClusterDims[] = {   hiRes.GetNumCells( 0 ) / loRes.GetNumCells( 0 )
                                            ,   hiRes.GetNumCells( 1 ) / loRes.GetNumCells( 1 )
                                            ,   hiRes.GetNumCells( 2 ) / loRes.GetNumCells( 2 ) } ;

            const unsigned  numPointsLoRes[3]   = { loRes.GetNumPoints( 0 ) , loRes.GetNumPoints( 1 ) , loRes.GetNumPoints( 2 ) };
            const unsigned  numXYLoRes          = loRes.GetNumPoints( 0 ) * loRes.GetNumPoints( 1 ) ;
            const unsigned  numPointsHiRes[3]   = { hiRes.GetNumPoints( 0 ) , hiRes.GetNumPoints( 1 ) , hiRes.GetNumPoints( 2 ) };
            const unsigned  idxShifts[3]        = { pClusterDims[0] / 2 , pClusterDims[1] / 2 , pClusterDims[2] / 2 } ;

            // Since this loop iterates over each destination cell, it parallelizes without contention.
            unsigned idxLoRes[3] ;
            for( idxLoRes[2] = unsigned( izStart ) ; idxLoRes[2] < unsigned( izEnd ) ; ++ idxLoRes[2] )
            {
                const unsigned offsetLoZ = idxLoRes[2] * numXYLoRes ;
                for( idxLoRes[1] = 0 ; idxLoRes[1] < numPointsLoRes[1] ; ++ idxLoRes[1] )
                {
                    const unsigned offsetLoYZ = idxLoRes[1] * loRes.GetNumPoints( 0 ) + offsetLoZ ;
                    for( idxLoRes[0] = 0 ; idxLoRes[0] < numPointsLoRes[0] ; ++ idxLoRes[0] )
                    {   // For each cell in the loRes layer...
                        const unsigned  offsetLoXYZ   = idxLoRes[0] + offsetLoYZ ;
                        ItemT        &  rValLoRes  = loRes[ offsetLoXYZ ] ;
                        unsigned clusterMinIndices[ 3 ] ;
                        unsigned idxHiRes[3] ;

                        if( UniformGridGeometry::FASTER_LESS_ACCURATE == accuracyVsSpeed )
                        {
                            memset( & rValLoRes , 0 , sizeof( rValLoRes ) ) ;
                            NestedGrid<ItemT>::GetChildClusterMinCornerIndex( clusterMinIndices , pClusterDims , idxLoRes ) ;
                            idxHiRes[2] = clusterMinIndices[2] ;
                            idxHiRes[1] = clusterMinIndices[1] ;
                            idxHiRes[0] = clusterMinIndices[0] ;
                            const unsigned offsetZ      = idxHiRes[2] * numXYhiRes ;
                            const unsigned offsetYZ     = idxHiRes[1] * numXhiRes + offsetZ ;
                            const unsigned offsetXYZ    = idxHiRes[0] + offsetYZ ;
                            const ItemT &  rValHiRes    = hiRes[ offsetXYZ ] ;
                            rValLoRes = rValHiRes ;
                        }
                        else
                        { ... see archive for full code listing...
                        }
                    }
                }
            }
        }
```

These routines have interesting twist compared to others in this series: They are methods of a templated class. That means the functor class must also be templated. The syntax is much easier when the functor class is nested within the UniformGrid class. Then, the fact that it is templated is implicit – the syntax is formally identical to a non-templated class.

Here is the functor class for DownSample. Note that it is defined inside the UniformGrid templated class:

```
    class UniformGrid_DownSample_TBB
    {
                  UniformGrid &                         mLoResDst               ;
            const UniformGrid &                         mHiResSrc               ;
            UniformGridGeometry::AccuracyVersusSpeedE   mAccuracyVersusSpeed    ;
        public:
            void operator() ( const tbb::blocked_range<size_t> & r ) const
            {   // Perform subset of down-sampling
                SetFloatingPointControlWord( mMasterThreadFloatingPointControlWord ) ;
                SetMmxControlStatusRegister( mMasterThreadMmxControlStatusRegister ) ;
                mLoResDst.DownSampleSlice( mHiResSrc , mAccuracyVersusSpeed , r.begin() , r.end() ) ;
            }
            UniformGrid_DownSample_TBB( UniformGrid & loResDst , const UniformGrid & hiResSrc
                                      , UniformGridGeometry::AccuracyVersusSpeedE accuracyVsSpeed )
                : mLoResDst( loResDst )
                , mHiResSrc( hiResSrc )
                , mAccuracyVersusSpeed( accuracyVsSpeed )
            {
                mMasterThreadFloatingPointControlWord = GetFloatingPointControlWord() ;
                mMasterThreadMmxControlStatusRegister = GetMmxControlStatusRegister() ;
            }
        private:
            WORD        mMasterThreadFloatingPointControlWord   ;
            unsigned    mMasterThreadMmxControlStatusRegister   ;
    } ;
```

Here is the wrapper routine for DownSample:

```
  void DownSample( const UniformGrid & hiResSrc , AccuracyVersusSpeedE accuracyVsSpeed )
  {
      const size_t numZ = GetNumPoints( 2 ) ;
# if USE_TBB
```

```
        {
            // Estimate grain size based on size of problem and number of processors.
            const size_t grainSize =  Max2( size_t( 1 ) , numZ / gNumberOfProcessors ) ;
            parallel_for( tbb::blocked_range<size_t>( 0 , numZ , grainSize )
                        , UniformGrid_DownSample_TBB( * this , hiResSrc , accuracyVsSpeed ) ) ;
        }
# else
        DownSampleSlice( hiResSrc , accuracyVsSpeed , 0 , numZ ) ;
# endif
    }
```

Performance

The table shows the duration (in milliseconds per frame) of various routines, run on a 3.50 GHz Intel Core i7-3770K that has 4 physical cores and 2 local cores per physical core.

# threads	Frame	Vorton Sim	Vector Potential	UpSample	Poisson	Render
1	29.8	6.64	2.37	0.0554	0.222	13.8
2	17.9	5.11	1.36	0.0205	0.148	7.53
3	13.1	4.69	1.28	0.0196	0.153	4.99
4	13.0	4.55	1.22	0.0116	0.148	5.04
8	11.1	4.44	1.13	0.0023	0.141	3.97

Notice that VortonSim does not speed up linearly with the number of cores. Perhaps the algorithms have reached the point where data access (not instructions) is the bottleneck.

Summary and Options

This article presented a fluid simulation that combines integral and differential numerical techniques to achieve an algorithm that takes time linear in the number of grid points or particles. The overall simulation can't be faster than that because each particle has to be accessed to be rendered. It also provides better results than the treecode because the latter uses approximations everywhere in the computational domain that the Poisson solver does not, and the Poisson solver has an inherently smoother and more globally accurate solution.

More work could be done to improve this algorithm: Currently, the numerical routines are broken up logically so that they are easier to understand. But this causes the computer to revisit the same data repeatedly. After data-parallelizing the routines, their run times become bound by memory access instead of instructions. So if instead all the fluid simulation operations were consolidated into a single monolithic routine that accessed the data only once, and that super-routine were parallelized, it might lead to even greater speed.

About the Author

Dr. Michael J. Gourlay works at Microsoft as a principal development lead on HoloLens in the Environment Understanding group.

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* and original architect of FranTk (the engine behind Connected Careers mode), on character physics and ANT (the procedural animation system used by EA), on *Mixed Martial Arts*, and as a lead programmer on *NASCAR.* He wrote Lynx (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, a top-rated 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. Michael received his degrees in physics and philosophy from Georgia Tech and the University of Colorado at Boulder.