Fluid Simulation for Video Games (part 16)

By Dr. Michael J. Gourlay

Vortex Particle Method Meets Smoothed Particle Hydrodynamics

The Vortex Particle Method (VPM) and Smoothed Particle Hydrodynamics (SPH) are both fluid simulation techniques that use particles to represent the fluid. Most articles in this series focus on VPM, which performs especially well for gaseous fluids because VPM retains filamentary detail without dissipating it. Part 15 describes a rudimentary SPH fluid simulation used to model a fluid in a container. SPH has a simpler boundary condition and so is well suited to viscous fluids in containers. This article—the 16th in the series—explores an example of how to apply SPH formulae to VPM. It also demonstrates that VPM and SPH readily complement each other because both operate on fundamentally similar principles, making it easy to alternate between them, even within the same simulation.

Part 1 in this series 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 between bodies and fluid. Part 11 added *combustion,* a chemical reaction that generates heat. 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, and part 14 modified those polytopes to represent rudimentary containers.

Mesh-free Gradients

Previously, the simulations that accompanied these articles computed density gradient by transferring density to a uniform grid, then computing gradients using finite differences. Uniform grids make that computation fast and simple, but they have uniform resolution even where the fluid might have nonuniformly distributed features (see Figure 1).

Instead, you can use an SPH methodology to estimate density gradients. Because SPH computes gradients using pairs of particles, it avoids the resolution disparity of using a uniform grid.

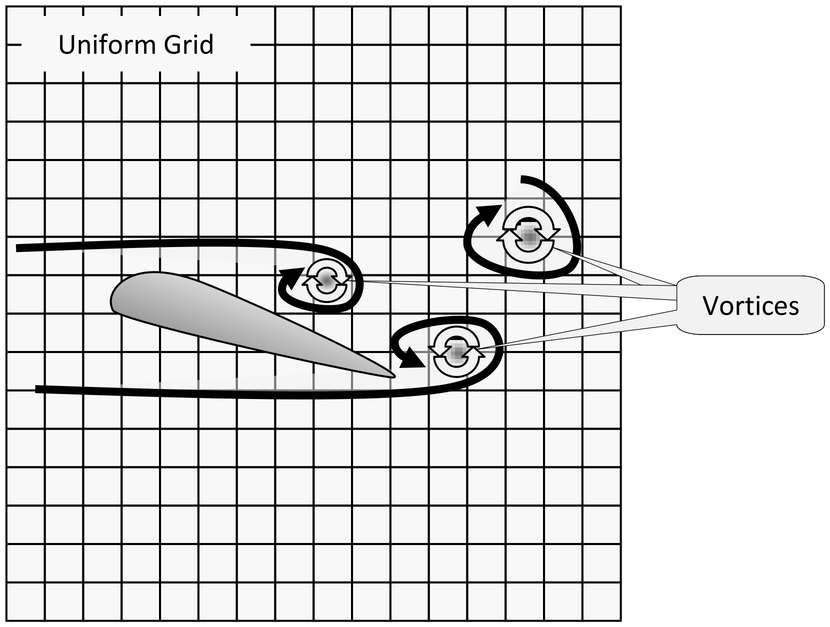


Figure 1. Uniform grid with nonuniformly distributed vortices.

Remember from part 15 that there are multiple SPH formulae for computing gradients, including *canonical* and *difference*. The canonical formula only uses local density values:

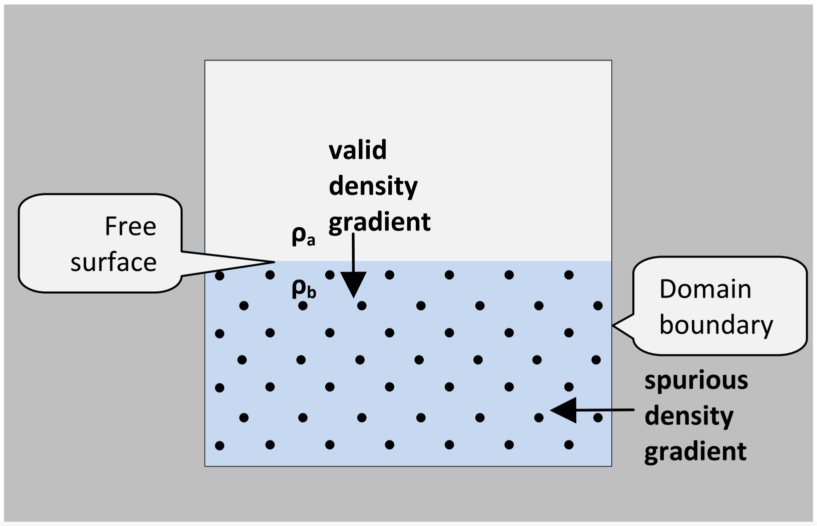
$$\nabla A(\vec{r}_j) = \sum_i \frac{A_i}{\rho_i} \nabla w(\vec{r}_{ij})$$

As in part 15, in these formulae, A is the quantity whose gradient you want, ρ_i is the number density of particle i, w is the smoothing kernel, the subscript *j* means "the value for the particle at $\vec{r}_j$," and $\vec{r}_{ij}$ is the separation between two particles, $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$.

Near walls, use per-particle mass density instead of smoothed density, and use a difference formula:

$$\nabla A(\vec{r}_j) = \frac{1}{\rho_j} \sum_i (A_i - A_j) \nabla w(\vec{r}_{ij})$$

Figure 2 shows a comparison between the two cases.



Both cases seem to have a density gradient due to the interface between particles and no particles.

But you only want the free surface (not the domain boundary) to have a density gradient.

Figure 2. Two cases that could lead to estimating a non-zero density gradient. You only want one of them to allow baroclinic vorticity generation.

Part 14 (in the section, "Pressure at Walls") dealt with the same situation in an analogous way: Grid cells that touched boundaries were marked as such, and their density gradients were set to zero.

Implementation

The following code snippet shows the kernel for computing mass density gradient. This gradient is computed using the same SPH framework presented in part 15, in the "Visitation Algorithm" subsection. (Note the kernel previously computed *number* density gradient to compute a pressure gradient acceleration. In contrast, this code computes *mass* density gradient.)

```
{   // Compute mass density gradient.
    const float smoothGrad  = -3.0f * q2 * mInvInflRad ;

    // Take into account ambient density by subtracting it from massDensX below.
    // Note that when using the "difference" gradient formula, that cancels out.
    // Using this, difference gradient is zero at boundaries
    // between fluid and empty space.
    const float massDensA = mParticles[ idxA ].mDensity - mAmbientDensity ;
    const float massDensB = mParticles[ idxB ].mDensity - mAmbientDensity ;

    // Canonical SPH gradient formula.
    const float densGradA_can = massDensB * smoothGrad / numDensB ;
    const float densGradB_can = massDensA * smoothGrad / numDensA ;

    // Difference SPH gradient formula.
    const float common_dif      = ( massDensB - massDensA ) * smoothGrad ;
    const float densGradA_dif   =   common_dif / numDensA ;
    const float densGradB_dif   = - common_dif / numDensB ;

    // Use difference formula at body boundaries, canonical formula elsewhere.
    if( mProximities[ idxA ] >= mInfluenceRadius )
    {   // Far from wall: Use canonical formula.
        mDensityGradients[ idxA ] += densGradA_can * dir ;
    }
    else
    {   // Near or in wall: Use difference formula.
        mDensityGradients[ idxA ] += densGradA_dif * dir ;
    }

    if( mProximities[ idxB ] >= mInfluenceRadius )
    {   // Far from wall: Use canonical formula.
        mDensityGradients[ idxB ] -= densGradB_can * dir ;
    }
    else
    {   // Near or in wall: Use difference formula.
        mDensityGradients[ idxB ] -= densGradB_dif * dir ;
    }
}
```

Note the use of `mProximities` in this code: Those are the signed distances to a wall. This routine computes those values:

```
void FluidBodySim::ComputeParticleProximityToWalls( VECTOR< float > & proximities , const VECTOR< Particle > & particles
                                   , const VECTOR< Impulsion::PhysicalObject * > & physObjs , const float maxProximity )
{
    const size_t numPcls = particles.Size() ;

    proximities.Clear() ;
    proximities.Resize( numPcls , maxProximity * ( 1.0f + FLT_EPSILON ) ) ;

    const size_t numPhysObjs = physObjs.Size() ;
    for( unsigned idxPhysObj = 0 ; idxPhysObj < numPhysObjs ; ++ idxPhysObj )
    {   // For each body...
        Impulsion::PhysicalObject &     physObj         = * physObjs[ idxPhysObj ] ;
        const Vec3 &                    physObjPosition = physObj.GetBody()->GetPosition() ;
        const Collision::ShapeBase *    collisionShape  = physObj.GetCollisionShape() ;
        const float &                   boundingRadius  = collisionShape->GetBoundingSphereRadius() ;
        const Impulsion::RigidBody *    rigidBody       = physObj.GetBody() ;

        for( unsigned iPcl = 0 ; iPcl < numPcls ; ++ iPcl )
        {   // For each particle...
            const Vec3 & pclPos = particles[ iPcl ].mPosition ;
            // Compute particle proximity to bodies.
            float & proximity = proximities[ iPcl ] ;

            const Vec3  vBodyCenterToParticle = pclPos - physObjPosition ;   // vector from body center to particle
            const float fBodyCenterToParticle = vBodyCenterToParticle.Magnitude() ;

            bool        broadPhaseCollision ;

            if( collisionShape->IsHole() )
            {
                const float combinedRadii  = Max2( boundingRadius - maxProximity , 0.0f ) ;
                broadPhaseCollision = fBodyCenterToParticle > combinedRadii ;
            }
            else
            {
                const float combinedRadii  = boundingRadius + maxProximity ;
                broadPhaseCollision = fBodyCenterToParticle < combinedRadii ;
            }

            if( broadPhaseCollision )
            {   // Particle is inside padded bounding sphere of rigid body.
                if( physObj.GetCollisionShape()->GetShapeType() == Collision::SphereShape::sShapeType )
                {   // Rigid body is a sphere, and gridpoint is inside its padded collision volume.
                    proximity = Min2( proximity , fBodyCenterToParticle ) ;
                }
                else if( physObj.GetCollisionShape()->GetShapeType() == Collision::ConvexPolytope::sShapeType )
                {   // Rigid body is a polytope.
```

```
                    // Test for collision with padded collision volume.
                    const Collision::ConvexPolytope *   convexPolytope      =
                        static_cast< const Collision::ConvexPolytope * >( collisionShape ) ;
                    const Mat33 &                       physObjOrientation  = rigidBody->GetOrientation() ;
                    unsigned                            idxPlane ;
                    const float                         contactDistance     =
                        convexPolytope->ContactDistance( pclPos , physObjPosition , physObjOrientation , idxPlane ) ;

                    if( contactDistance < maxProximity )
                    {   // Gridpoint is in contact with padded rigid body collision volume.
                        proximity = Min2( proximity , contactDistance ) ;
                    }
                }
            }
        }
    }
}
```

Parallelization

Intel® Threading Building Blocks (Intel® TBB) can speed up this process. The thread routine has the same stenciled form as the visitation routines presented in part 15 and has this signature:

```
static void ComputeSphMassDensityGradient_Grid_Slice( VECTOR< Vec3 > & densityGradients
                                     , const VECTOR< SphFluidDensities > & fluidDensitiesAtPcls
                                     , const VECTOR< Vorton > & particles
                                     , const VECTOR< float > & proximities
                                     , const UniformGrid< VECTOR< unsigned > > & pclIndicesGrid
                                     , const float ambientDensity
                                     , size_t izStart , size_t izEnd
                                     , VortonSim::PhaseE phase )
```

That gets called through a functor:

```
    class SphSim_ComputeSphPressureGradientAcceleration_TBB
    {
            VECTOR< Vec3 > &                            mDensityGradients       ;
            const VECTOR< SphFluidDensities > &         mFluidDensitiesAtPcls   ;
            const VECTOR< Vorton > &                    mParticles              ;
            const VECTOR< float > &                     mProximities            ;
            const UniformGrid< VECTOR< unsigned > > &   mPclIndicesGrid         ;
            const float                                 mAmbientDensity         ;
            VortonSim::PhaseE                           mPhase                  ;

        public:
            void operator() ( const tbb::blocked_range<size_t> & r ) const
            {   // Compute particle acceleration due to pressure gradients for subset of domain.
                SetFloatingPointControlWord( mMasterThreadFloatingPointControlWord ) ;
                SetMmxControlStatusRegister( mMasterThreadMmxControlStatusRegister ) ;
                ComputeSphDensityGradient_Grid_Slice(mDensityGradients
                    , mFluidDensitiesAtPcls , mParticles , mProximities , mPclIndicesGrid , mAmbientDensity
                    , r.begin() , r.end() , mPhase ) ;
            }

            SphSim_ComputeSphDensityGradient_TBB(
                , VECTOR< Vec3 > &                          densityGradients
                , const VECTOR< SphFluidDensities > &       fluidDensitiesAtPcls
                , const VECTOR< Vorton > &                  particles
                , const VECTOR< float > &                   proximities
                , const UniformGrid< VECTOR< unsigned > > & pclIndicesGrid
                , const float                               ambientDensity
                , VortonSim::PhaseE                         phase
                )
                , mDensityGradients( densityGradients )
                , mFluidDensitiesAtPcls( fluidDensitiesAtPcls )
                , mParticles( particles )
                , mProximities( proximities )
                , mPclIndicesGrid( pclIndicesGrid )
                , mAmbientDensity( ambientDensity )
                , mPhase( phase )
            {
                mMasterThreadFloatingPointControlWord = GetFloatingPointControlWord() ;
                mMasterThreadMmxControlStatusRegister = GetMmxControlStatusRegister() ;
            }
        private:
            WORD        mMasterThreadFloatingPointControlWord   ;
            unsigned    mMasterThreadMmxControlStatusRegister   ;
    } ;
```

This driver routine uses Intel® TBB to invoke the functor on multiple threads:

```
void ComputeSphDensityGradient_Grid( VECTOR< Vec3 > & densityGradients
                                    , const VECTOR< SphFluidDensities > & fluidDensitiesAtPcls
                                    , const VECTOR< Vorton > & particles
                                    , const VECTOR< float > & proximities
                                    , const UniformGrid< VECTOR< unsigned > > & pclIndicesGrid
                                    , const float ambientDensity )
{
    densityGradients.clear() ;
    densityGradients.resize( particles.Size() , Vec3( 0.0f , 0.0f , 0.0f ) ) ;

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

    #if USE_TBB
        // Estimate grain size based on size of problem and number of processors.
        const size_t grainSize =  Max2( size_t( 1 ) , nzm1 / gNumberOfProcessors ) ;
        // Compute particle mass density using threading building blocks.
        // Alternate between even and odd z-slices to avoid multiple threads accessing the same particles simultaneously.
        parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize ) ,
            SphSim_ComputeSphDensityGradient_TBB( accelerations , densityGradients , fluidDensitiesAtPcls
                , particles , proximities , pclIndicesGrid , ambientDensity , VortonSim::PHASE_EVEN ) ) ;
        parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize ) ,
            SphSim_ComputeSphDensityGradient_TBB( accelerations , densityGradients , fluidDensitiesAtPcls
                , particles , proximities , pclIndicesGrid , ambientDensity , VortonSim::PHASE_ODD  ) ) ;
    #else
        ComputeSphDensityGradient_Grid_Slice( accelerations , densityGradients , fluidDensitiesAtPcls , particles
            , proximities , pclIndicesGrid , ambientDensity , 0 , nzm1 , VortonSim::PHASE_BOTH ) ;
    #endif
}
```

These routines are used inside `VortonSim::GenerateBaroclinicVorticity`, originally described in part 3, now modified to use SPH instead of a grid:

```
        FluidBodySim::ComputeParticleProximityToWalls( mVortonBodyProximities
            , reinterpret_cast< VECTOR< Particle > & >( * mVortons ) , * mPhysicalObjects , inflRad ) ;
        ComputeSphPressureGradientAcceleration_Grid( accelerationOfPcls_DUMMY , mDensityGradientsAtPcls , mFluidDensitiesAtPcls
            , * mVortons , mVortonBodyProximities , vortonIndicesGridSph , mAmbientDensity ) ;
```

Mixing VPM and SPH

VPM and SPH have their relative merits. Both methods operate with the same overall form: Compute a velocity field, then advect particles according to that field. This lets you mix the methods.

Remember from part 15 that this implementation of SPH includes an interparticle force that tries to keep the particles a fixed distance apart. As a result, when mixing these methods, vortons tend to cling together.

Drop Falling in Tank

Figure 3 shows three approaches to the same scenario: a drop of fluid falling in a box.

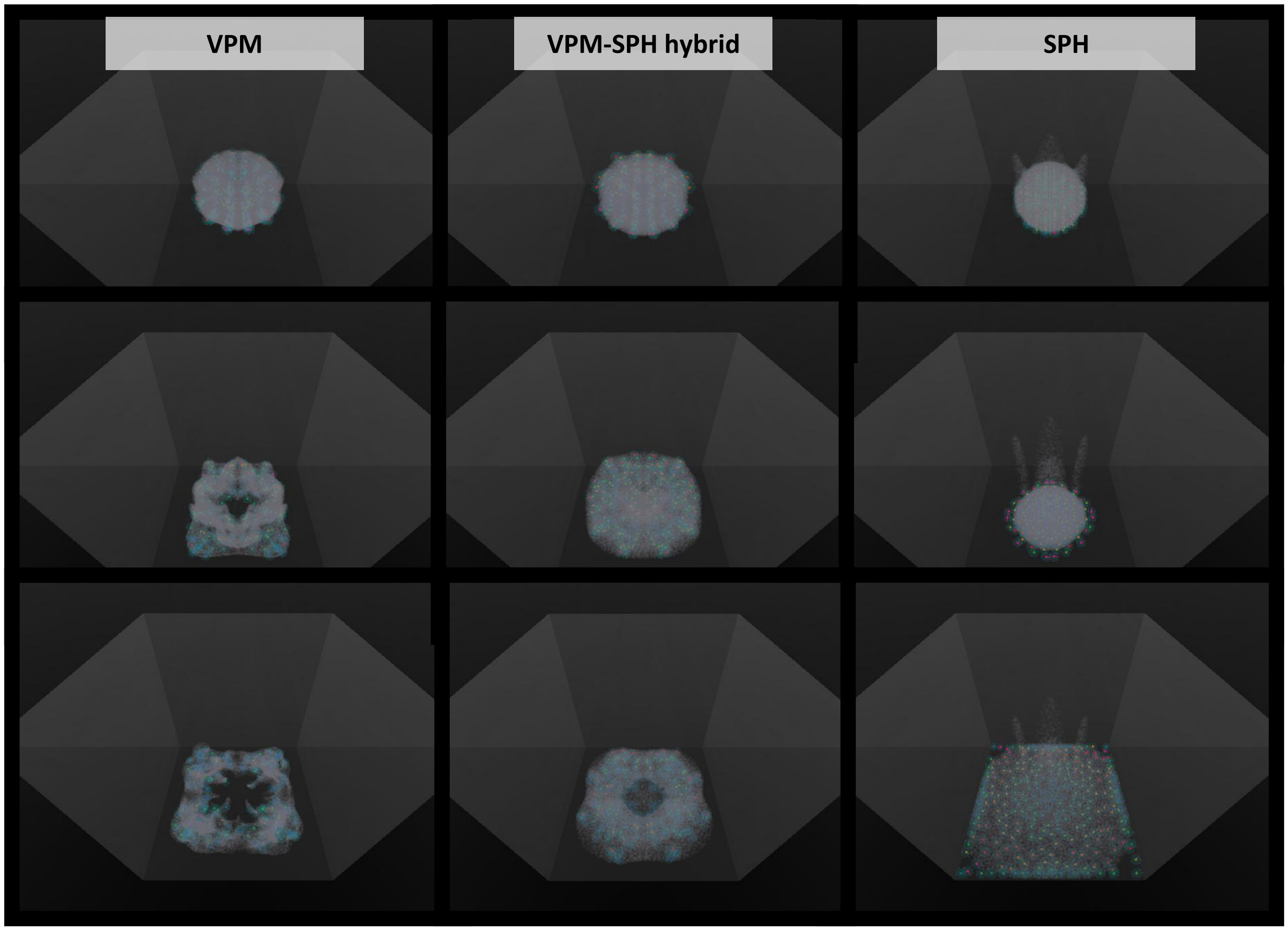


Figure 3. Oblique view of a drop falling using VPM, SPH, and hybrid simulation methods. Grey shows passive tracers. Cyan blobs show vortices, which have colored arrows showing vorticity.

Notice that the methods that use VPM result in the drop forming a vortex ring. Methods that use SPH tend to keep the drop together. The hybrid shows both behaviors.

Performance

The SPH formulation of baroclinic vorticity generation iterates through each particle in each grid cell to find its neighbors and applies the SPH gradient formula. In contrast, the grid-based algorithm iterates through each grid cell and uses neighboring grid points to compute density gradient. The SPH version has more conditional branches and iterates over more input items, so it is slow.

Table 1 shows runtimes for `VortonSim_GenerateBaroclinicVorticity` for two scenarios: a drop falling in free space (with no boundaries) and a drop falling inside a box. Remember that the density gradient is computed differently (for both grid and SPH versions) at and near boundaries, so the scenario with boundaries takes more time than the scenario without.

- Drop (no boundaries):
  - Grid based: 0.167206 ms per frame
  - SPH based: 0.515427 ms per frame
- Drop in box:
  - Grid based with poison: 0.255762 ms per frame
  - SPH based: 0.654715 ms per frame

Summary

Part 15 described the SPH methodology. This article showed how to use SPH, instead of a grid, to compute density gradient. The simulation accompanying this article uses that to compute baroclinic generation of vorticity. The result is mesh free and automatically dynamically adapts to nonuniform distributions of particles but takes more time to run.

Future Articles

The particle rendering used in the figures and accompanying code is poor for liquids. It would be better to render only the liquid surface. To do that, you would need a surface-tracking and extraction algorithm. You could use the same information to model surface tension.

Particle methods, including SPH and vortex, rely on spatial partitioning to accelerate neighbor searches. The uniform grid used in this series is simplistic and not the most efficient, and populating it takes more time than it should. It would be worthwhile to investigate various spatial partition algorithms to see which runs fastest, especially with the benefit of multiple threads.

Future articles will investigate these questions.

Further Reading

- Golia, C., Buonomo, B., & Viviani, A. (2009). Grid free Lagrangian blobs vortex method with Brinkman layer domain embedding approach for heterogeneous unsteady thermo fluid dynamics problems. *International Journal of Engineering (IJE), 3*(3).
  - Describes a vortex particle method that “penalizes” the vorticity of any vortex particles that approach or enter a body. The penalty term tends to make the vorticity satisfy boundary conditions. During their diffusive step, vortex particles in that “Brinkman layer” near the body walls do not have a baroclinic term. Their approach is not entirely unlike the one I adopt in part 15 and in this article.

About the Author

Dr. Michael J. Gourlay works as a senior software engineer on interactive entertainment. 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,* on character physics and the procedural animation system used by EA on *Mixed Martial Arts*, 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, Florida Interactive Entertainment Academy, 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 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.