Fluid Simulation for Video Games (part 13)

By Dr. Michael J. Gourlay

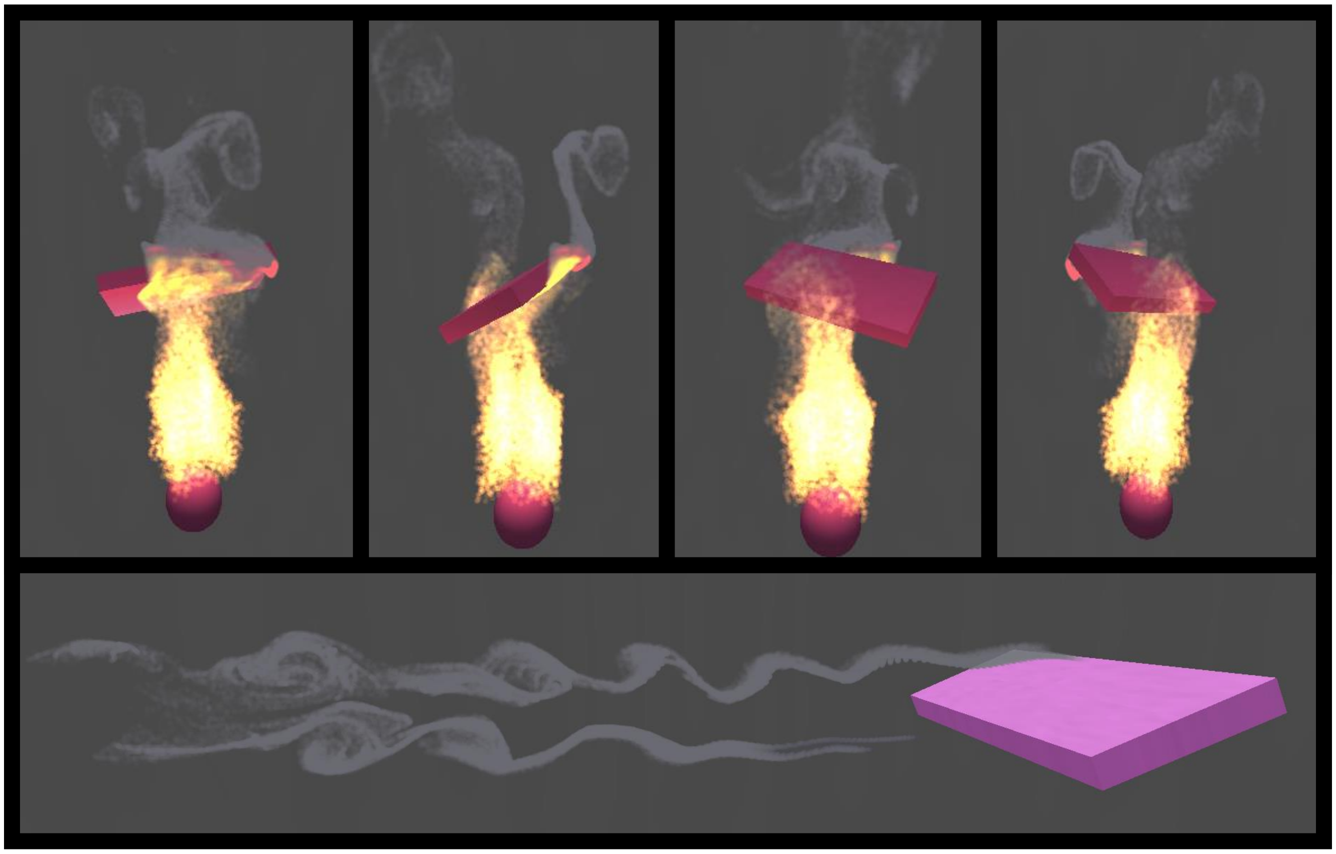

Figure 1. Convex polyhedra interacting with a vortex particle fluid

Convex Obstacles

Video games are compelling, because they are interactive. Even visual effects should respond to other entities in the environment, especially those the user controls. Particle effects, including fluids, should therefore respond to rigid bodies of any shape. Those shapes should include airfoils that can experience lift.

This article—the thirteenth in a series—describes how to augment the fluid particle system described earlier, interact with rigid bodies with any polyhedral shape, and generate a lift-like force on those bodies. 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. Part 12 explained how improper sampling caused unwanted jerky motion and described how to mitigate it.

Collision Detection

Detecting collisions between objects first entails computing the distance between them. Video games model most shapes with planar, polygonal faces and treat particles as though they have spherical shape. You therefore need to compute the distance between planes and a spheres.

The Math Behind Planes

A *plane* is a two-dimensional surface in a three-dimensional space. You can define a plane in various ways. One convenient representation uses a normal vector $\hat{n} \equiv \langle n_x, n_y, n_z \rangle$ and a distance, *d.* The *plane equation* has this formulation:

$$n_x\, x + n_y\, y + n_z\, z - d = 0$$

All points with coordinates $\langle x, y, z \rangle$ that satisfy this equation lie within the plane. When $\hat{n}$ has unit length ($|\hat{n}| = 1$), d is the distance of the plane from the origin. You can use this value to represent a plane as a vector with four components (that is, a 4-vector plane): $\langle n_x, n_y, n_z, d \rangle$ or $\langle \hat{n}, d \rangle$.

The distance, D, of a point $\vec{q} \equiv \langle x, y, z \rangle$ from the plane $\langle \hat{n}, d \rangle$ is:

$$D(\vec{q}, \langle \hat{n}, d \rangle) = n_x\, x + n_y\, y + n_z\, z - d$$

Notice that this equation has the same form as the equation for the plane, except that instead of equating it to zero, the formula tells you the distance of the point to the plane. This is obviously consistent, because if a point has zero distance to the plane, then the point lies in the plane.

But wait a moment! This formula could result in negative values. For example, choose the point $\vec{q} \equiv \langle 0,0,0 \rangle$ and the plane with $\hat{n} = \langle 1,0,0 \rangle$ and $d = 1$. The formula claims that point has *negative* distance from the plane. What the heck is *negative distance?*

A plane divides all of space into two halves. A *half-space* is the region of space on one side of a plane. The signed distance formula tells you whether a point lies in the positive half-space or the negative half-space of a plane, as Figure 2 depicts.

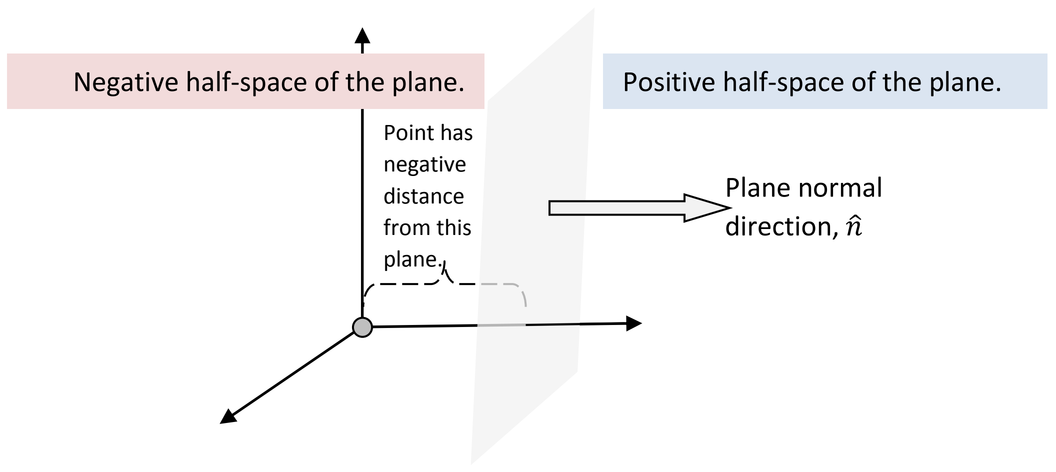
Negative half-space of the plane.
Positive half-space of the plane.
Point has negative distance from this plane.
Plane normal direction, $\hat{n}$

Figure 2. Planes and half-spaces

Think of a plane as facing in the direction of its normal. Points behind the plane have negative distance. Points in front of the plane have positive distance. When a point lies behind a plane, we call its distance (which is negative) the *penetration depth.*

The `Plane` class shows an implementation of the planar formulae that builds upon a 4-vector class (`Vec4`):

```
class Plane : private Vec4
{
    public:
        /** Construct plane representation from floats.
        */
        Plane( const Vec3 & normal , const float distFromOrigin )
        {
            x = normal.x ; y = normal.y ; z = normal.z ;
            w = distFromOrigin ;
        }


        Plane( const Vec4 & v4 )
        {
            x = v4.x ; y = v4.y ; z = v4.z ; w = v4.w ;
        }


        const Vec3 & GetNormal() const
        { return reinterpret_cast< const Vec3 & >( * this ) ; }


        float Distance( const Vec3 & vPoint ) const
        {
            const float distFromPlane =
                vPoint.operator*( reinterpret_cast< const Vec3 & >( * this ) ) - w ;
            return distFromPlane ;
            }
} ;
```

You can compute the distance of a sphere from a plane by computing the distance of the sphere's center (a point) from the plane, then subtracting the sphere's radius.

Note: Technically, planes exist in other dimensions. For example, in a two-dimensional space, a plane is also a line. In a four-dimensional space, a plane is a three-dimensional hyperplane. But this article discusses three-dimensional spaces, where planes are two dimensional.

Planes Make Convex Hulls

A *polytope* is a shape with flat sides. In two dimensions, polytopes are called *polygons.* In three dimensions, polytopes are called *polyhedra.*

A *convex shape* is one where any line segment between any two points in the shape is also in that shape. So, a *convex polytope* is such a shape with flat sides, as shown in Figure 3.

An array of planes can represent a convex polyhedron: Describe each face, *i,* of the polyhedron using a plane representation, $\langle \hat{n}_i, d_i \rangle$. This is called a *half-space representation* (or *H-representation*).

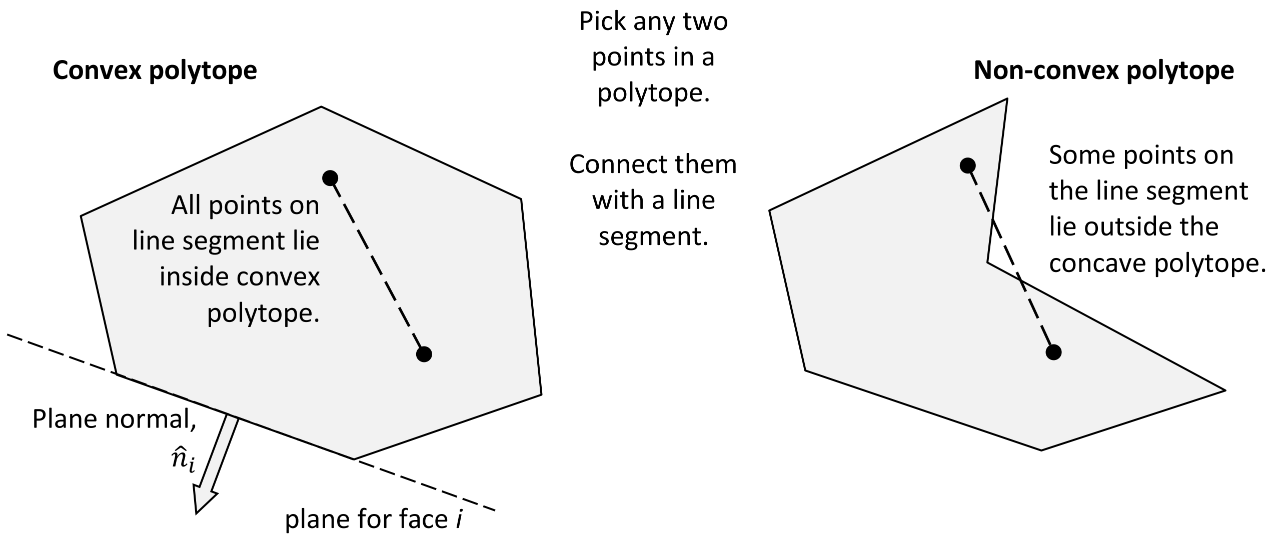
Convex polytope
All points on line segment lie inside convex polytope.
Plane normal, $\hat{n}_i$
plane for face *i*
Pick any two points in a polytope.
Connect them with a line segment.
Non-convex polytope
Some points on the line segment lie outside the concave polytope.

Figure 3. Convex versus non-convex polytopes

To determine whether a point is inside a polyhedron, compute the distance of that point to each face plane of the polyhedron. If all distances are negative, the point lies inside the polyhedron.

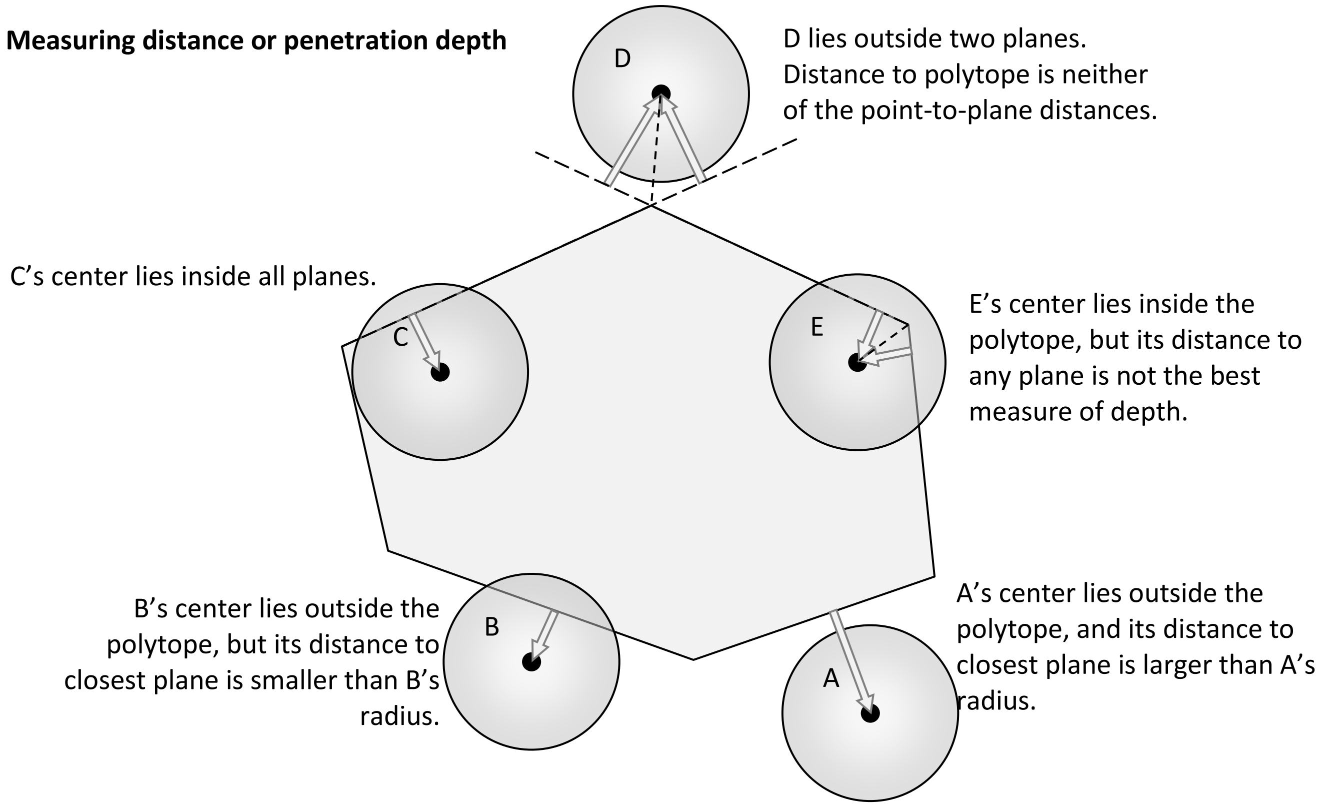
Measuring distance or penetration depth
D
D lies outside two planes. Distance to polytope is neither of the point-to-plane distances.
C's center lies inside all planes.
C
E
E's center lies inside the polytope, but its distance to any plane is not the best measure of depth.
B's center lies outside the polytope, but its distance to closest plane is smaller than B's radius.
B
A
A's center lies outside the polytope, and its distance to closest plane is larger than A's radius.

Figure 4. Measuring distance or penetration depth between stationary spheres and polytopes

As Figure 4 depicts, computing the distance between a stationary point (or sphere) and the planes of a polytope does not always give an unambiguous measure of distance or penetration depth. Sometimes, the best measure of distance could be from an edge or vertex of the polytope. But the distance formula will always correctly tell whether a point (or sphere) is inside, outside, or overlapping a polytope.

Alternative Method

The point-to-plane method suffices when detecting collisions between particles and polytopes. Other algorithms exist to compute the distance between two convex shapes. One of the most famous and useful, especially among game developers, is the Gilbert–Johnson–Keerthi (GJK) distance algorithm. Although its code is fast and simple, the concepts are not familiar nor easy enough to explain to fit in this article. Furthermore, determining penetration depth can be even more problematic and usually entails more sophisticated approaches, such as the Expanding Polytope Algorithm (EPA). See the "For Further Study" section at the end of this article for more information.

Collision Detection

Detecting a collision between objects entails computing their separation distance or penetration depth. Also, when objects collide, you usually want to know the region of contact and *contact normals*—that is, the direction along which to apply force or displacement to separate the objects.

Although the point-to-plane distance formula will tell you whether a point lies inside a polytope, it will not unambiguously tell you its distance or penetration depth. In addition to the edge cases described earlier, determining penetration depth entails the relative direction of travel of the two objects. The correct answer depends on the configuration of objects before and after the collision. If objects lie inside each other (or if a particle lies inside a polytope), then you have detected the collision after it occurred. This is called *interpenetration,* and it should be avoided or corrected when it happens.

For visual effects involving hundreds or thousands of tiny particles, you can get adequate results by using the following simple algorithm:

```
1. Given a query point, a polytope, its position and orientation, initialize largestDistance to
   an extremely large, negative value.
2. For each plane in the polytope:
   a. Compute the distance between the query point and the plane.
   b. If that distance exceeds largestDistance, then:
      i.   Assign largestDistance to that distance
      ii.  Remember this plane index
3. Return the plane index and largestDistance.
```

For spheres, subtract their radius from the returned largest distance to get the separation distance. If that value is negative, the sphere interpenetrates the polytope.

From that information, you can compute a contact point and normal:

```
Given a query point, a plane, a polytope orientation, and the largest distance:
1. Reorient the plane normal to world space.
2. Scale the normal for the returned plane index by the largest distance to get a penetration
   vector.
3. Subtract the penetration vector from the query point to get the contact point.
```

Although this algorithm does not accurately measure distance for the edge cases, the distance and normal it returns yield sufficiently close results to work for collision response.

The `ConvexPolytope` class implements these algorithms. (See the demonstration code that accompanies these articles for more details.)

```
float ConvexPolytope::ContactDistance( const Vec3 & queryPoint ,
    unsigned & idxPlaneLeastPenetration )
{
    float largestDistance = - FLT_MAX ;
    const size_t numPlanes = mPlanes.Size() ;
    for( size_t iPlane = 0 ; iPlane < numPlanes ; ++ iPlane )
    {   // For each planar face of this convex hull...
        const float distToPlane = mPlanes[ iPlane ].Distance( reorientedPoint ) ;
        if( distToPlane > largestDistance )
        {   // Point distance to iPlane is largest of all planes visited so far.
            largestDistance = distToPlane ;
            idxPlaneLeastPenetration = iPlane ;
        }
    }
    return largestDistance ;
}
```

The routine `ContactPoint` uses information computed by `ContactDistance`:

```
Vec3 ConvexPolytope::ContactPoint( const Vec3 & queryPoint , const Mat33 & orientation
    , const unsigned idxPlaneLeastPenetration , const float distance
    , Vec3 & contactNormal )
{
    const Math::Plane & originalPlane = mPlanes[ idxPlaneLeastPenetration ] ;
    contactNormal                     = orientation.Transform( originalPlane.GetNormal() ) ;
    Vec3                contactPoint  = queryPoint - contactNormal * distance ;
    return contactPoint ;
}
```

Broad Phase

The `ContactDistance` algorithm iterates over every face in a polyhedron. That process can get expensive. You can reduce that expense in a few ways:

- Only compute `ContactDistance` when the query point lies within a coarse bounding volume (such as a bounding sphere) that contains the polytope. That computation is much faster and can let you skip the more expensive `ContactDistance` until the query point is somewhat near the polytope. The accompanying code uses this technique.
- If you only care about interpenetration, you can change `ContactDistance` to return immediately when it finds any distance-to-plane that is positive. The returned value will not necessarily be the largest distance, but when positive, you don't care. Note that if you want to compute the distance to a sphere instead of to a point, then you would have to pass in the sphere radius and take that into account. The accompanying code includes a routine that uses this technique.

You could get fancy and reduce CPU cost further at the expense of memory and complexity:

- Remember the plane from the previous iteration, and reuse that for the next attempt. If it's still positive, there is no collision, and you can bail out after testing one plane. Note that this would entail storing another integer per particle. Because there can be tens of thousands of particles, that can add up.

- Store face connectivity information, and only visit adjacent faces whose distance would increase `largestDistance`. Doing so can significantly reduce the number of faces visited. Also, game engines often include such adjacency information. You might be able to exploit that information for particle collisions.

Deepening Penetration

If a particle penetrates an object, it could end up closer to the opposite side of the object rather than the side it penetrated, as Figure 5 shows. This could happen for thin objects or fast particles. It is therefore useful to consider only those planes for which particles are moving farther behind.

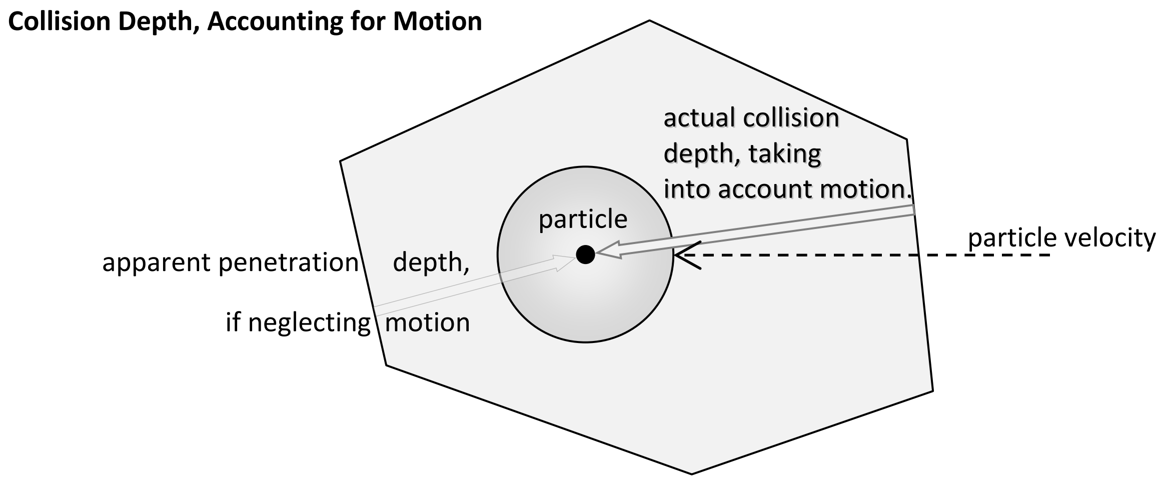
Collision Depth, Accounting for Motion
actual collision depth, taking into account motion.
particle
particle velocity
apparent penetration depth, if neglecting motion

Figure 5. Measuring collision depth between a moving spherical particle and polytope

The demo code accompanying this article contains a routine, `ConvexPolytope::CollisionDistance`, that implements this idea.

Continuous Collision Detection

The most accurate way to determine contacts would entail *continuous collision detection* (CCD)—that is, detecting the collision just as it happens (instead of after the fact). CCD involves computing a *time of impact* (TOI) and either advancing the simulation up to that point or rewinding back to it. One way to approximate CCD to estimate TOI is to move into a reference frame where only the object moves. The other object will still be in motion. Now, *sweep* that moving object across space to span the region it would occupy at all points during the test interval. If the swept shape intersects with the stationary shape, the two objects probably collided during that interval.

Sweeping a shape is relatively easy if its motion is pure translation but more difficult if its motion includes rotation. Video games therefore either treat only linear motion for continuous collision detection or simply use discrete collision detection and allow objects to interpenetrate.

For spherical shapes like particles, the swept shape is a line segment with hemispherical caps, also known as a *capsule* or *sausage.* You can compute intersections between capsules and planes using simple formulae. But particle effects for video games do not need that level of sophistication, and it takes longer to compute than most games budget for effects.

Concave Shapes

The technique described in this article applies directly to convex shapes. To apply to concave shapes, you can either compute the convex hull of that shape or decompose the shape into convex components. See the "For Further Study" section for more information.

Collision Response

When the detection phase indicates that a particle interpenetrated an obstacle, the simulation must resolve the collision. In other words, it must push the particle outside the obstacle and adjust the fluid flow to satisfy boundary conditions.

Part 1, part 2, and part 4 explained boundary conditions and one way to solve them approximately, so I will not repeat that here. This article only describes changes that facilitate particles interacting with convex polyhedra.

Simplified Vorton Interaction with Planes

The routine `SolveBoundaryConditions` iterates through each rigid body and collides vortons and tracers with that body by calling `CollideVortonsSlice` and `CollideTracersSlice`. As of this article, `ColideVortonsSlice` is a new routine, extracted from `SolveBoundaryConditions` from previous articles.

The code snippet below focuses on changes made to facilitate colliding with convex polytopes. Code in **purple bold** is new.

```
void FluidBodySim::CollideVortonsSlice( Vector< Particle > & particles
    , float ambientFluidDensity , float fluidSpecificHeatCapacity
    , const Impulsion::PhysicalObject & physObj , Vec3 & rLinearImpulseOnBody
    , Vec3 & rAngularImpulseOnBody , float & rHeatToBody , size_t iPclStart , size_t iPclEnd )
{
    rLinearImpulseOnBody    = Vec3( 0.0f , 0.0f , 0.0f ) ;
    rAngularImpulseOnBody   = Vec3( 0.0f , 0.0f , 0.0f ) ;
    rHeatToBody              = 0.0f ;

    // Collide vortons with rigid body.
    for( size_t uVorton = iPclStart ; uVorton < iPclEnd ; ++ uVorton )
    {   // For each vorton in the simulation...
        Vorton &      rVorton               = static_cast< Vorton & >( particles[ uVorton ] ) ;
        const float   vortRadius            = rVorton.GetRadius() ;
        const float   vortRadius2           = vortRadius * vortRadius ;
        const Vec3    vSphereToVorton       = rVorton.mPosition - physObj.GetBody()->GetPosition();
        const float   fSphereToVorton       = vSphereToVorton.Magnitude() ;
        const Vec3    vSphereToVortonDir    = vSphereToVorton / fSphereToVorton ;
        const float   fBndThkFactor       = 1.2f ; // Thickness of boundary, in vorton radii.
        const float   fBoundaryThickness  = fBndThkFactor * vortRadius ;
        const float & physObjRadius = physObj.GetCollisionShape()->GetBoundingSphereRadius();

        // "Contact" point, near where vorton touched body.
        Vec3 vContactPtRelBody  ;
        Vec3 vContactPtWorld    ;

        if( fSphereToVorton < ( physObjRadius + fBoundaryThickness ) )
        {   // Vorton lies within bounding sphere of rigid body.
            if( physObj.GetCollisionShape()->GetShapeType() == Collision::SphereShape::sShapeType )
            {   // Rigid body is a sphere, and vorton is inside it.
                vContactPtRelBody   = vSphereToVortonDir * physObjRadius ;
                vContactPtWorld     = vContactPtRelBody + physObj.GetBody()->GetPosition() ;
            }
            else
            {   // Rigid body is a polytope.
                const Collision::ConvexPolytope *   convexPolytope      =
                    static_cast<const Collision::ConvexPolytope *  >( physObj.GetCollisionShape() );
                const Vec3 &   physObjPosition     = physObj.GetBody()->GetPosition() ;
                const Mat33 &  physObjOrientation  = physObj.GetBody()->GetOrientation() ;
                size_t         idxPlane ;
                const float    contactDistance     = convexPolytope->ContactDistance(
                    rVorton.mPosition , physObjPosition , physObjOrientation , idxPlane ) ;

                if( contactDistance < rVorton.GetRadius() )
                {   // Tracer is in contact rigid body.
                    // Compute contact point.
                    Vec3       contactNormal ;
                    vContactPtWorld     = convexPolytope->ContactPoint(
                        rVorton.mPosition , physObjOrientation , idxPlane , contactDistance
                      , contactNormal ) ;
                    vContactPtRelBody   = vContactPtWorld - physObj.GetBody()->GetPosition();
                }
                else
                {   // Vorton is NOT in contact with rigid body.
                    continue ;  // Skip to next vorton.
                }
            }
            // (omitted) ...Eject vorton from body....
            // (omitted) ...Compute vorticity change to satisfy boundary conditions....
            // (omitted) ...Compute heat transfer between fluid and body....
            // (omitted) ...Compute angular momentum transfer between fluid and body....
        }
    }
}
```

Notice that this code first checks whether the particle lies within a bounding sphere, regardless of whether the obstacle is a sphere or polytope. That is a broad-phase collision test.

The routines `CollideTracersSlice` and `RemoveEmbeddedParticles` have similar changes. See the demonstration code accompanying this article for details.

Parallelization

The routine `CollideVortonsSlice` was extracted from `SolveBoundaryConditions` to facilitate parallelizing it with Intel® Threading Building Blocks (Intel® TBB). In addition to extracting that code into its own routine, other changes were made. Previously, the corresponding code directly applied changes to the rigid body's temperature and momentum. The old code performed operations like this:

1. Read body temperature.
2. Compute heat exchange based on body temperature.
3. Write new body temperature.

But when run in parallel, such updates would cause a race condition, as shown in Table 1.

Table 1. Parallel threads cause a race condition.

Thread 1	Thread 2
Read body temperature.	Read body temperature.
Compute heat exchange based on body temperature.	Compute heat exchange based on body temperature.
–	Write new body temperature.
Write new body temperature.	–

Both threads would update a value at the same address (temperature, in this example). Only one thread can "win."

You could solve this issue by synchronizing the code with mutex locks on the body temperature. But doing so would serialize that critical section of code, which would in turn defeat the purpose of parallelizing it.

Instead, have each thread accumulate changes in a variable local to each thread. When the thread terminates, have the parent thread accumulate those changes and apply them to the body. This might seem to be a perfect use case for Intel® TBB's `parallel_reduce` operation.

There is one more catch, however: That accumulation operation is not associative. (See Part 12 for details of a similar problem.) Even though addition is associative for real numbers, it is not for floating-point numbers. To ensure that this parallelized routine is deterministic, you have to spawn and join threads manually, because Intel® TBB's `parallel_reduce` does not split and join deterministically. Instead, use Intel® TBB's `parallel_invoke` and manually spawn and join threads.

```
void FluidBodySim::CollideVortonsReduce( Vector< Particle > & particles
    , float ambientFluidDensity , float fluidSpecificHeatCapacity
    , const Impulsion::PhysicalObject & physObj , Vec3 & rLinearImpulseOnBody
    , Vec3 & rAngularImpulseOnBody , float & rHeatOnBody , size_t iPclStart , size_t iPclEnd
    , size_t grainSize )
{
    const size_t indexSpan = iPclEnd - iPclStart ;
    if( indexSpan <= grainSize )
    {   // Sub-problem fits into a single serial chunk.
        CollideVortonsSlice( particles , ambientFluidDensity , fluidSpecificHeatCapacity
            , physObj , rLinearImpulseOnBody , rAngularImpulseOnBody , rHeatOnBody
            , iPclStart , iPclEnd ) ;
    }
    else
    {   // Problem remains large enough to split into pieces.
```

```
        size_t  iPclMiddle = iPclStart + indexSpan / 2 ;
        // Create one functor for each sub-problem.
        FluidBodySim_CollideVortons_TBB cv1( particles , ambientFluidDensity
            , fluidSpecificHeatCapacity , physObj , iPclStart  , iPclMiddle , grainSize ) ;
        FluidBodySim_CollideVortons_TBB cv2( particles , ambientFluidDensity
            , fluidSpecificHeatCapacity , physObj , iPclMiddle , iPclEnd    , grainSize ) ;
        // Invoke both sub-problems, each on a separate thread.
        tbb::parallel_invoke( cv1 , cv2 ) ;
        // Combine results from each thread.
        rLinearImpulseOnBody    = cv1.mLinearImpulseOnBody  + cv2.mLinearImpulseOnBody  ;
        rAngularImpulseOnBody   = cv1.mAngularImpulseOnBody + cv2.mAngularImpulseOnBody ;
        rHeatOnBody             = cv1.mHeatToBody           + cv2.mHeatToBody           ;
    }
}
```

Create a functor to run `CollideVortonsReduce` with Intel® TBB `parallel_invoke`:

```
class FluidBodySim_CollideVortons_TBB
{ /// Function object to collide vortons (vortex particles) with rigid bodies.
        Vector< Particle > &                mVortons                    ;
        const Impulsion::PhysicalObject &   mPhysicalObject             ;
        float                               mAmbientFluidDensity        ;
        float                               mFluidSpecificHeatCapacity  ;

        WORD        mMasterThreadFloatingPointControlWord ; ///< FPCW from spawning thread.
        unsigned    mMasterThreadMmxControlStatusRegister ; ///< MXCSR from spawning thread.

        size_t      mBegin      ;   ///< Loop start for use with parallel_reduce.
        size_t      mEnd        ;   ///< Loop end for use with parallel_reduce.
        size_t      mGrainSize  ;   ///< Target number of elements to process per thread.

    public:
        /// Constructor to use with parallel_invoke.
        FluidBodySim_CollideVortons_TBB( Vector< Particle > & rVortons
            , float ambientFluidDensity , float fluidSpecificHeatCapacity
            , const Impulsion::PhysicalObject & physObj , size_t begin , size_t end
            , size_t grainSize )
            : mVortons( rVortons )
            , mAmbientFluidDensity( ambientFluidDensity )
            , mFluidSpecificHeatCapacity( fluidSpecificHeatCapacity )
            , mPhysicalObject( physObj )
            , mBegin( begin )
            , mEnd( end )
            , mGrainSize( grainSize )
            , mLinearImpulseOnBody( 0.0f , 0.0f , 0.0f )
            , mAngularImpulseOnBody( 0.0f , 0.0f , 0.0f )
            , mHeatToBody( 0.0f )
        {
            mMasterThreadFloatingPointControlWord = GetFloatingPointControlWord() ;
            mMasterThreadMmxControlStatusRegister = GetMmxControlStatusRegister() ;
        }

        /// Invocation operator for use with parallel_invoke.
        void operator() () const
        {   // Compute collisions for a subset of tracers.
            SetFloatingPointControlWord( mMasterThreadFloatingPointControlWord ) ;
            SetMmxControlStatusRegister( mMasterThreadMmxControlStatusRegister ) ;
            FluidBodySim::CollideVortonsReduce( mVortons , mAmbientFluidDensity
                , mFluidSpecificHeatCapacity , mPhysicalObject , mLinearImpulseOnBody
                , mAngularImpulseOnBody , mHeatToBody , mBegin , mEnd , mGrainSize ) ;
        }

        // Thread-local storage:
        mutable Vec3    mLinearImpulseOnBody    ;
        mutable Vec3    mAngularImpulseOnBody   ;
        mutable float   mHeatToBody             ;
} ;
```

For comparison, the demonstration code accompanying this article also includes code for using Intel® TBB's `parallel_reduce`. It works—in the sense that it generates a usable result—but it is not deterministic, so using it would impede diagnosing issues.

Results

Let's replace some of the spheres in previous articles with polytopes.

Although the demonstration code uses boxes, the algorithms and data structures support any convex polytope, as depicted later in Figure 8.

Scenarios

Figure 6 shows a flat plate interacting with flames and smoke. Notice that particles move around the plate, and the plate causes vortices to shed from it.

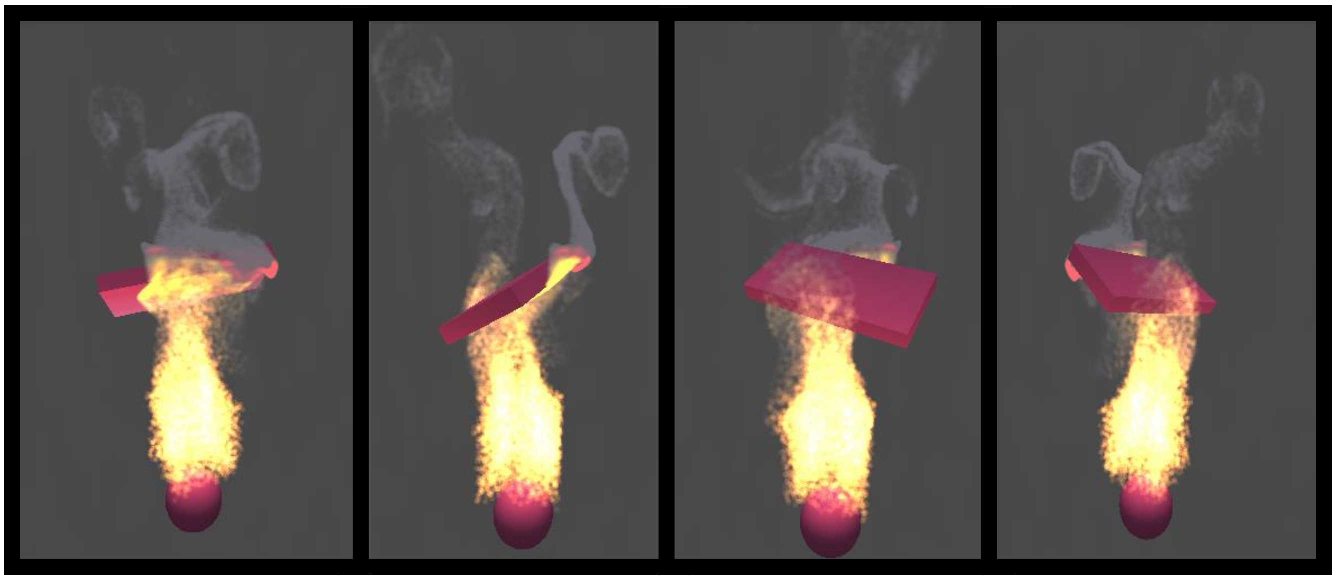

Figure 6. Various views of a plate above flames

Figure 7 shows a flat plate moving horizontally through fluid, leaving a wake with vortices. The code accompanying this article also includes demonstrations with the obstacle rotating about longitudinal and lateral axes, exhibiting the Magnus (curve ball) effect.

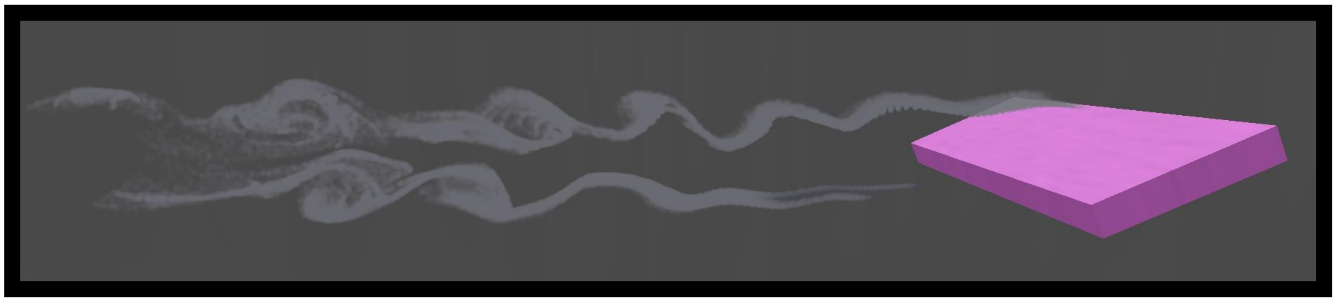

Figure 7. Flat plate moving through fluid

Figure 8 shows a polyhedral airfoil moving horizontally through fluid, leaving a wake with vortices. This demonstrates that the technique applies to shapes other than the boxes and spheres.

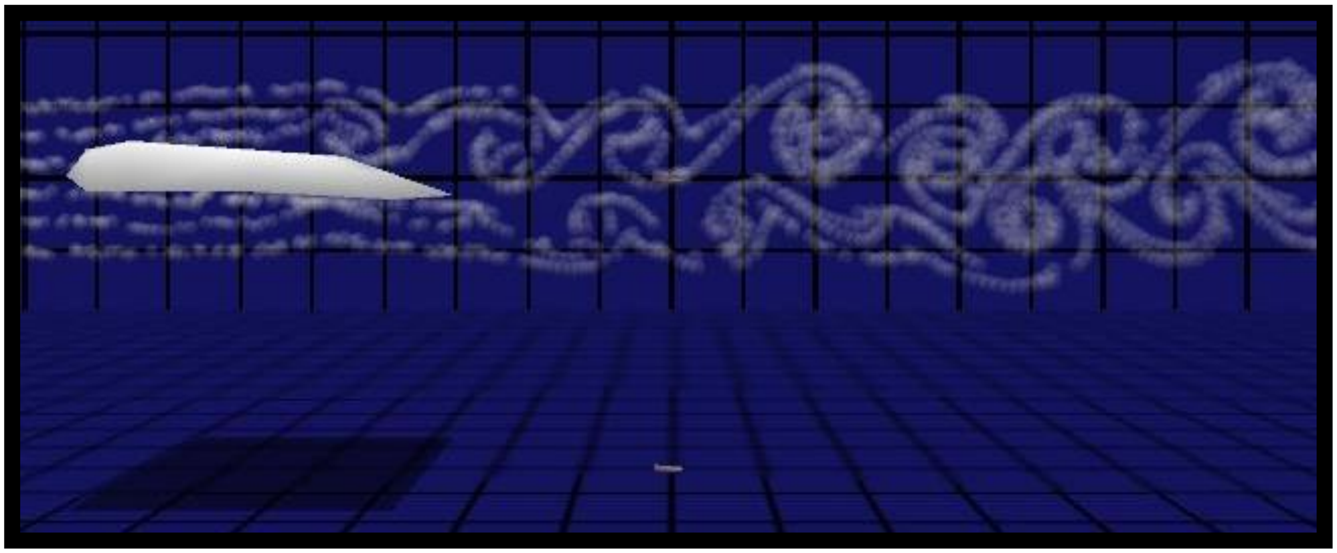

Figure 8. Airfoil moving through fluid. This comes from Gourlay (2010), which used a similar formulation to that presented in this article.

Performance

Table 2 shows how the collision-detection and response routines perform for the scenario with flames and smoke passing by the flat plate. This scenario had, on average, 49,000 tracer particles and 981 vortex particles (per frame), two spheres, and one box. Tracers hit bodies 8876 times per frame, and vortons hit bodies 63 times per frame. The benchmark ran 6000 frames for each run. The processor was a four-core (eight hardware threads with hyperthreading) Intel® Core™ i7-2600 running at 3.4 GHz.

Table 2. Collision-detection and response routines for the smoke and flame scenario

No. of threads	Solve boundary conditions	SBC tracers	SBC vortons	Sim update	Total (including render)
1	2.42	2.346	0.057	4.12	25.7
2	1.606	1.563	0.0423	2.52	16.3
3	1.134	1.095	0.0378	2.15	11.7
4	1.142	1.107	0.0348	2.03	11.4
6	0.804	0.774	0.0312	1.81	9.65
8	0.784	0.75	0.0303	1.76	9.34
12	0.69	0.657	0.0303	1.61	8.38
16	0.718	0.684	0.0336	1.64	8.59

Figure 9 shows a plot of the data in the table.

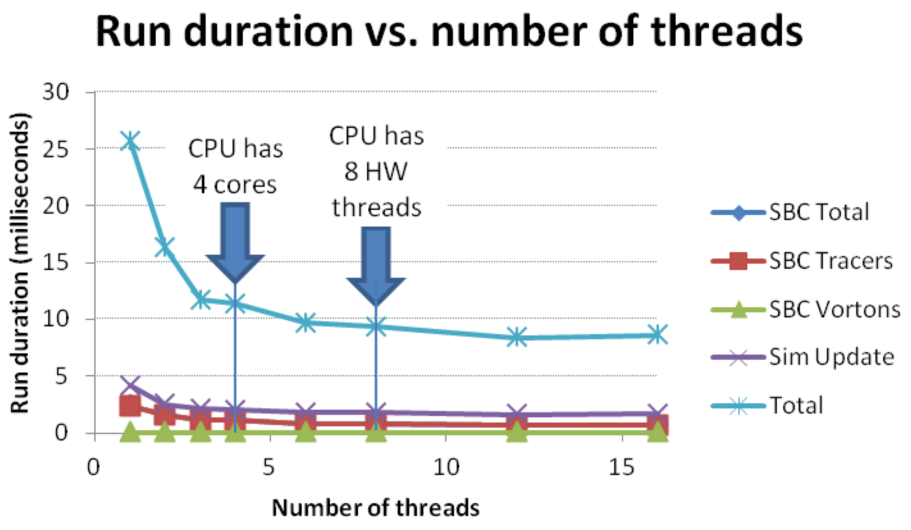

Figure 9. Run times for the benchmark scenario

Lift

Nonrotating spheres do not generate lift. But lift occurs on asymmetric shapes like flat plates and airfoils.

The algorithm to solve boundary conditions generates lift-like impulses. A flat plate moving horizontally through the fluid at an appropriate angle of attack should encounter lift pushing the plate upward. And indeed, this simulation generates a qualitatively similar result—but it's from deflecting particles that bounce off the obstacle with partially elastic collisions. In contrast, simulations used in science and engineering calculate pressure the fluid exerts on bodies, and that calculation can include lift. This simulation does not calculate pressure.

Furthermore, the collision response algorithm does not generate new vortons; it only reassigns values for existing vortons in contact with the object. To be more physically accurate, objects should generate new vortons when necessary. For example, vorticity would be generated on the leeward side of the airfoil, and this would generate a low-pressure region behind and above the airfoil, the vertical component of which would be lift. See the "For Further Study" section for more information about more physically accurate ways to calculate realistic pressure and aerodynamic forces on objects immersed in a fluid.

Summary

Using simple a point-to-plane distance formula, you can make fluid effects interact with obstacles that have shapes commonly used to create models in a video game. The algorithm is easy to parallelize using Intel® TBB and runs in less than a millisecond for tens of thousands of particles.

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 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.

For Further Study

- Casey Muratori posted a video (https://mollyrocket.com/849) that explains the GJK algorithm using straightforward geometry.
- Lien & Amato ("Approximate Convex Decomposition of Polyhedra," 2006) describe an algorithm to decompose a concave model into nearly convex shapes. Lien's Ph.D. dissertation (http://cs.gmu.edu/~jmlien/masc/uploads/Main/lien-dissertation.pdf) contains pseudo-code for their algorithms and a comprehensive bibliography on the subject. Also see their technical report (http://cs.gmu.edu/~jmlien/masc/uploads/Main/cd3d_TR_2006.pdf).
- The Wikipedia article on Convex Hull Algorithms (http://en.wikipedia.org/wiki/Convex_hull_algorithms) describes algorithms to obtain the convex hull of a set of points, such as the vertices of a model.

- In chapter 6 of *Vortex Methods: Theory and Practice*, Cottet & Koumoutsakos describe a vorticity creation algorithm to satisfy boundary conditions. In contrast to the algorithm presented here and in part 4, theirs takes into account the entire body at once rather than only a single point at a time.
- I presented a formulation similar to that described in this article in "Fluid-body simulation using vortex particle operations" in the animation poster session at the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Los Angeles, in 2010.

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 Boulder.