Fluid Simulation for Video Games (part 10)

By Dr. Michael J. Gourlay

Figure 1. Hot yellow ball heats fluid with red dye, which then expands and rises.

"Hot Air Rises"—well, sort of.

Heat transfers by contact during collisions at the molecular level, and air expands when it heats. The expanded air occupies more volume but has the same mass, therefore has lower density. Lower density air buoys above higher density air. Thus, hot air rises.

A previous article (Part 8) already explained how a vortex-based fluid simulation can handle variable density in a fluid. So to get hot air to rise, the simulation must incorporate thermal effects: conduction, diffusion and expansion.

This article—the tenth in a series—describes how density varies with temperature, how heat transfers throughout a fluid, and how heat transfers between bodies and fluid. Part 1 summarized fluid dynamics; part 2 surveyed fluid simulation techniques, and 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 9 described how to approximate buoyant and gravitational forces on a body immersed in a fluid with varying density.

This article introduces features to simulation code presented in previous articles: Now, bodies immersed within the fluid heat or cool surrounding fluid (and vice versa), fluid can transfer heat within itself, and fluid density depends on its temperature. These new features augment how visual effects have a two-way interaction with physical objects in the simulation, and will segue into the next article, on combustion, in which a chemical process will heat fluid.

Thermal Effects

As mentioned in part 8, this simulation employs the Boussinesq approximation, in order to neglect compressibility effects -- essentially, skipping sound waves, which are computationally expensive to simulate, and they do not usually affect fluid flow enough to worry about them. Although it assumes fluid does not compress much due to pressure, the Boussinesq approximation does allow fluid density to change according to its temperature.

Note: As mentioned in part 8, the Boussinesq approximation has many limitations. To learn about improving that approximation, read about the anelastic and pseudo-incompressible approximations. See Further Reading.

Thermal Expansion

As mentioned above, hot air rises because that air is less dense. The simulation needs to quantify how density relates to temperature.

Fluid density, ρ_{fluid}, depends on temperature, T, according to this formula:

$$\frac{\delta\rho_{fluid}}{\rho_{fluid}} = -\alpha\,\delta T$$

Relative change in density	equals	negative thermal expansivity times temperature change

or

$$\frac{\rho_0 - \rho}{\rho_0} = \alpha(T - T_0)$$

where ρ_0 is a reference density when the fluid is at some reference temperature T_0. Solving this for ρ leads to $\rho = \rho_0[1 - \alpha(T - T_0)]$. Here, α is the coefficient of thermal expansion, defined as

$$\alpha \equiv \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_p = \frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_p$$

Thermal expansion describes how a material's volume changes with temperature, as depicted in Figure 2. Note that this formula is a definition (it's not derived from anything), and that α is in general a function of temperature (not a constant).

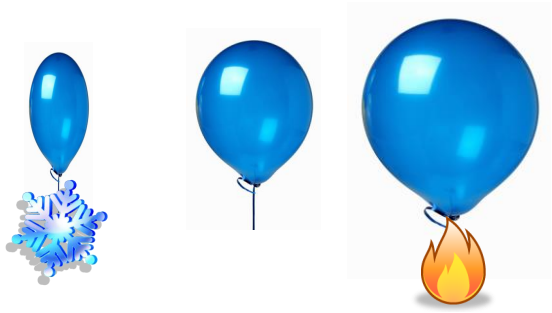

Figure 2. Thermal contraction and expansion of a gas.

To derive a formula for thermal expansion coefficient for an ideal gas, apply that formula to the ideal gas law,

$$pV = nRT,$$

(where n is the amount of substance and R is the universal ideal gas constant) to get α=$1/T$. Combined, this leads to the approximation $\rho = \rho_0 T_0 / T$. For our simulation code, we use the ambient fluid density for ρ_0. (See part 8 for details about this so-called ambient density.)

This formula lets the simulation reuse density instead of having to keep track of fluid temperature as a separate property; temperature is a simple function of density, and vice versa. Later in the article, we'll see how to implement this in the simulation code.

Thermal diffusion

The fluid simulation presented so far has concerned itself with mechanical energy. Temperature, in contrast, is tied to the "internal energy", such as vibration and rotation of molecules. So now the simulation needs an equation that expresses what happens to internal energy when temperature varies in a fluid. In particular, we need to know how heat moves around, even when the fluid does not.

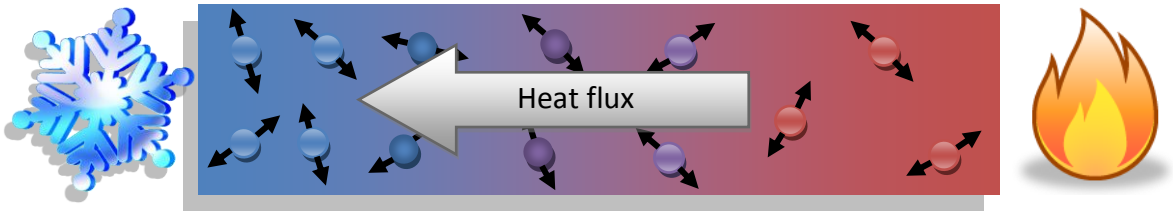

Figure 3. Heat diffuses along a temperature gradient. Dots with arrows depict molecules moving around at a microscopic scale.

The internal energy equation (a.k.a thermal energy equation, a.k.a. heat equation) for a fluid describes how heat varies with time and space:

$$\rho \frac{De}{Dt} = -\vec{\nabla} \cdot \vec{q} - p(\vec{\nabla} \cdot \vec{u}) + \varphi$$

Internal energy increase	equals	heat convergence	volume compression	viscous dissipation

Here, e is the internal energy and $\vec{q}$ is the heat flux, that is, the rate at which heat enters and leaves a tiny volume. And φ is viscous heating: When fluid molecules collide, they lose some of their kinetic energy to internal energy, in the form of molecular vibration, rotation and translation, also known as **heat**.

Note: The internal energy equation comes from the first law of thermodynamics, which is an expression of conservation of energy. See Further Reading for details, including derivations and explanations of this formula.

Given the thermal expansion formula from above in the form $(D\rho/DT)_p = -\alpha\rho$, we can modify the volume compression term, $-p(\vec{\nabla} \cdot \vec{u})$, as

$$-p(\vec{\nabla} \cdot \vec{u}) = \frac{p}{\rho}\frac{Dp}{Dt} \approx \frac{p}{\rho}\left(\frac{D\rho}{DT}\right)_p \frac{DT}{Dt} = -p\,\alpha\frac{DT}{Dt}$$

Assume the fluid is an ideal gas with equation of state $p = \rho RT$. (We can use different equations of state for liquids, but for now assume an ideal gas.) With judicious application of Maxwell relations, this also implies $C_p - C_V = R$, where C_p is specific heat capacity at constant pressure and C_V is specific heat capacity at constant volume. (See, for example, Mandl p. 125). As mentioned above, $\alpha = 1/T$. Also, for an ideal gas, $e = C_V T$. The internal energy equation for an ideal gas becomes:

$$-p\vec{\nabla} \cdot \vec{u} = -\rho R\, T\, \alpha\frac{DT}{Dt} = -p(C_p - C_V)\frac{DT}{Dt}$$

Plug this back into internal energy equation to get

$$\rho\, C_p \frac{DT}{Dt} = -\vec{\nabla} \cdot \vec{q} + \varphi$$

Viscous dissipation is important for the mechanical energy equation, and having φ in both places (with opposite signs) shows that kinetic energy lost equals thermal energy gained. (Since φ is always positive, heat energy flows only one way -- from kinetic to thermal.) But for fluid velocities well below the speed of sound, we can neglect its contribution to heat. So drop φ. This leaves the terms $\rho C_p \frac{DT}{Dt} = -\vec{\nabla} \cdot \vec{q}$.

Note: Viscous dissipation is proportional to the square of velocity gradients, which can also be expressed as the mean of vorticity dotted with itself, which is called **enstrophy**. See, it all comes back to vorticity!

Fourier's law of heat conduction

To get any use out of the internal energy equation, we need an expression for heat flux, $\vec{q}$.

Within the material depicted in Figure 3, the molecules move around, jostling other molecules they contact. Each collision transfers internal energy, and this is called **conduction**.

Fourier's law of heat conduction states that the heat transfers in the opposite direction of the temperature gradient (that is, from hot to cold):

$$\vec{q} = -k\nabla T$$

Here, k is conductivity (where $k = \kappa\rho C_p$) and ∇T is the temperature gradient. Plug that into the internal energy equation above to get a diffusion equation:

$$\frac{DT}{Dt} = \kappa\nabla^2 T.$$

We've seen this equation before -- back in part 3 -- except for a vector (vorticity) instead of a scalar (temperature). Naively computing that using the Laplacian (∇^2) would require computing second-order spatial derivatives and do a lot of bookkeeping that would complicate the simulation code. But the simulation will employ a trick -- particle strength exchange -- to simplify the computation. See "*Computing thermal diffusion within the fluid*" below.

Note: Since the Boussinesq approximation relates temperature to density, heat diffusion implies mass diffusion. This implies mass diffuses like the way salinity (or some other solute) would. Read up on Fick's laws of diffusion for more info about how diffusion relates to flux.

Adding Thermal Expansion and Diffusion to the Simulation

To add the effects of thermal expansion and diffusion, the simulation must compute them.

The simulation code requires these modifications:

A. Treat vortex particles (vortons) as carriers of heat for the fluid.
B. Compute thermal diffusion within the fluid, using particle strength exchange.
C. Compute thermal conduction between rigid bodies and the fluid.

Computing thermal expansion

From the modifications made in part 8, vortons already carry density for the fluid.

The thermal expansion formula derived above lets the simulation reuse density instead of having to keep track of fluid temperature as a separate property; temperature is a simple function of density, and vice versa. Add the following methods to the Particle class:

```
        float GetTemperature( float ambientDensity ) const
        {
            return /* ambientTemperature* */ambientDensity / ( ambientDensity + mDensity ) ;
        }
        void SetTemperature( float ambientDensity , float temperature )
        {
            float density = ambientDensity /* *ambientTemperature */ / temperature ;
            mDensity = density - ambientDensity ;
        }
```

Simplify this a step further by assuming that ambientTemperature is 1 in whatever units the simulation uses, and omit that factor from the formulas in the code above.

Technically, not even this change is necessary since we can compute thermal diffusion and conduction directly using density, but it's illustrative to have these formulae in code.

Remember from part 8 that the baroclinic generation of vorticity causes fluid with density lower than its surroundings to rise. So by simply lowering density, the fluid will rise "automatically".

Computing thermal diffusion within the fluid

As described above, this partial differential equation describes how heat diffuses through a fluid:

$$\frac{DT}{Dt} = \kappa \nabla^2 T.$$

As described in part 3, the simulation can compute diffusion by exchanging heat between nearest neighboring particles -- so-called "particle strength exchange" (PSE). Doing so efficiently requires a fast spatial partition to find nearby vortons. The simulation already creates that partition for vortex diffusion, so we can reuse it for thermal energy diffusion.

To reuse the vorton partition, first extract out the spatial partitioning into its own routine. This isn't new code; it's just in a new place:

```
void VortonSim::PartitionVortons( const float & timeStep , const unsigned & uFrame
  , UniformGrid< Vector< unsigned > > & ugVortonIndices )
{
    const size_t numVortons = mVortons->Size() ;
    for( unsigned offset = 0 /* Start at 0th vorton */ ; offset < numVortons ; ++ offset )
    {   // For each vorton...
        Vorton &    rVorton         = (*mVortons)[ offset ] ;
        // Insert the vorton's offset into the spatial partition.
        ugVortonIndices[ rVorton.mPosition ].PushBack( offset ) ;
    }
}
```

The simulation (VortonSim::Update, to be precise) calls PartitionVortons before calling DiffuseVorticityPSE.

Next, the simulation needs to exchange heat, which it does in a routine called DiffuseHeatPSE. The recipe follows exactly that used for DiffuseVorticityPSE. The code is so similar, in fact, that this

article will only show the "innards" of the routine -- which gets called inside an inner loop that is otherwise identical to that used to diffuse vorticity. (Of course, the accompanying code contains everything, so see it for details.)

```
void VortonSim::ExchangeHeat( const unsigned & rVortIdxHere , Vorton & rVortonHere
  , float & rDensityHere , const unsigned & ivThere , const Vector< unsigned > & cell
  , const float & timeStep )
{
    const unsigned &    rVortIdxThere   = cell[ ivThere ] ;
    Vorton &            rVortonThere    = (*mVortons)[ rVortIdxThere ] ;
    float &             rDensityThere   = rVortonThere.mDensity ;
    const float         densityDiff     = rDensityHere - rDensityThere ;
    const float         exchange        = 2.0f * mThermalDiffusivity * timeStep * densityDiff
;
    rDensityHere  -= exchange ;   // Make "here"  temperature a little closer to "there".
    rDensityThere += exchange ;   // Make "there" temperature a little closer to "here".
}
```

Note that this routine directly exchanges density (not temperature). For an ideal gas, whose thermal expansion coefficient varies with temperature, this is technically improper; the quantity being exchanged is inversely proportional to temperature. But the result qualitatively the same -- particles exchange head until they reach equilibrium. Furthermore, this formulation is correct for materials that have constant thermal expansion coefficients. Either way, for visual effects for video games, the distinction is negligible.

Computing thermal conduction between bodies and fluid

To transfer heat between rigid bodies and the fluid, add some code to the simulation that handles collisions between vortons and bodies and exchange heat analogous to how the simulation transfers angular momentum between vortons and bodies. This code goes inside FluidBodySim::SolveBoundaryConditions:

```
// Conduct heat from body to fluid.
const float vortonTemperatureOld        = rVorton.GetTemperature( ambientFluidDensity ) ;
const float temperatureDifference       = rSphere.mTemperature - vortonTemperatureOld ;
const float heatConduction           = temperatureDifference * rSphere.mThermalConductivity;
const float heatExchange                = heatConduction /* * timeStep */ ;
const float vortonHeatCapacity = 1.0f ;
const float vortonTemperatureNew = vortonTemperatureOld + heatExchange * vortonHeatCapacity ;
rVorton.SetTemperature( ambientFluidDensity , vortonTemperatureNew ) ;
// Conduct heat from fluid to body.
rSphere.mTemperature += heatExchange * rSphere.mOneOverHeatCapacity ;
```

Note that this routine takes a few shortcuts: the timestep is ignored, and the heat capacity for a vorton is independent of its size. The rationale is that, for these simulations, the timestep is fixed, and the vorton size is uniform and constant, so their effects are incorporated into mThermalConductivity. The code shows, in comments, where those values would be incorporated if for some reason you wanted to have variable timesteps or vorton size -- but beware that such a change would likely require additional changes elsewhere.

Using Intel® Threading Building Blocks™ to Speed Up the Simulation

Computing DiffuseHeatPSE costs some CPU time (see below to see exactly how much). To mitigate that cost, use Intel® Threading Building Blocks™ (TBB) to parallelize it.

Since the heat diffusion uses basically the same algorithm as for vorticity diffusion, simply reuse the same recipe used to parallelize vortex diffusion:

(1) Write heat diffusion such that it can operate on arbitrary "slices" of data. In this case, slice the data across vortons:

```
void VortonSim::DiffuseHeatPSESlice( const float & timeStep
    , const UniformGrid< Vector< unsigned > > & ugVortonIndices
    , size_t izStart , size_t izEnd )
{   // Exchange heat with nearest neighbors:
    // For each cell in the spatial partition...
    //     For each vorton in each cell...
    ExchangeHeat( ... ) ;
}
```

(2) Create a functor to compute heat diffusion:

```
class VortonSim_DiffuseHeatPSE_TBB
{
        float         mTimeStep       ;   ///< Duration since last time step.
        VortonSim *   mVortonSim      ;   ///< Address of VortonSim object
        const UniformGrid< Vector< unsigned > > &   mUgVortonIndices ;
    public:
        void operator() ( const tbb::blocked_range<size_t> & r ) const
        {   // Compute subset of heat diffusion.
            mVortonSim->DiffuseHeatPSESlice(mTimeStep,mUgVortonIndices,r.begin(),r.end());
        }
        VortonSim_DiffuseHeatPSE_TBB( float timeStep , VortonSim * pVortonSim
                                    , const UniformGrid< Vector< unsigned > > & ugVortonIndices )
            : mTimeStep( timeStep )
            , mVortonSim( pVortonSim )
            , mUgVortonIndices( ugVortonIndices )
        {}
} ;
```

(3) Use Intel® Threading Building Blocks™ (TBB) to invoke the functor:

```
void VortonSim::DiffuseHeatPSE( const float & timeStep , const unsigned & uFrame
    , const UniformGrid< Vector< unsigned > > & ugVortonIndices )
{   // Exchange heat with nearest neighbors
    const unsigned & nz     = ugVortonIndices.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( 1 , nzm1 / gNumberOfProcessors ) ;
        // Compute heat diffusion using threading building blocks
        parallel_for( tbb::blocked_range<size_t>( 0 , nzm1 , grainSize )
```

```
                    , VortonSim_DiffuseHeatPSE_TBB( timeStep , this , ugVortonIndices ) ) ;
    #else
        DiffuseHeatPSESlice( timeStep , ugVortonIndices , 0 , nzm1 ) ;
    #endif
#endif
}
```

Results

Figure 4 shows a simulation with a hot ball embedded inside a fluid with temperature-sensitive dye. Initially the dye is blue, indicating cold fluid. As the ball gradually heats the fluid, and the heat diffuses through the fluid, the dye turns red.

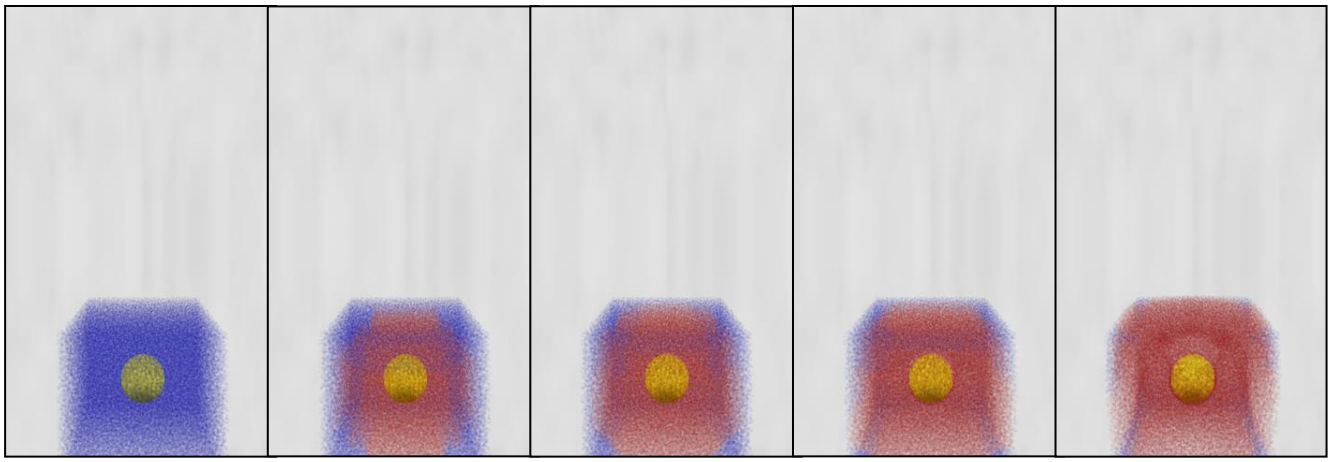

Figure 4. Thermal conduction and diffusion. The warm ball heats the surrounding fluid, and then heat diffuses throughout the fluid.

Figure 5 shows a continuation of the sequence in Figure 4: The heated fluid expands (so its density decreases) and therefore buoyancy causes that fluid to rise.

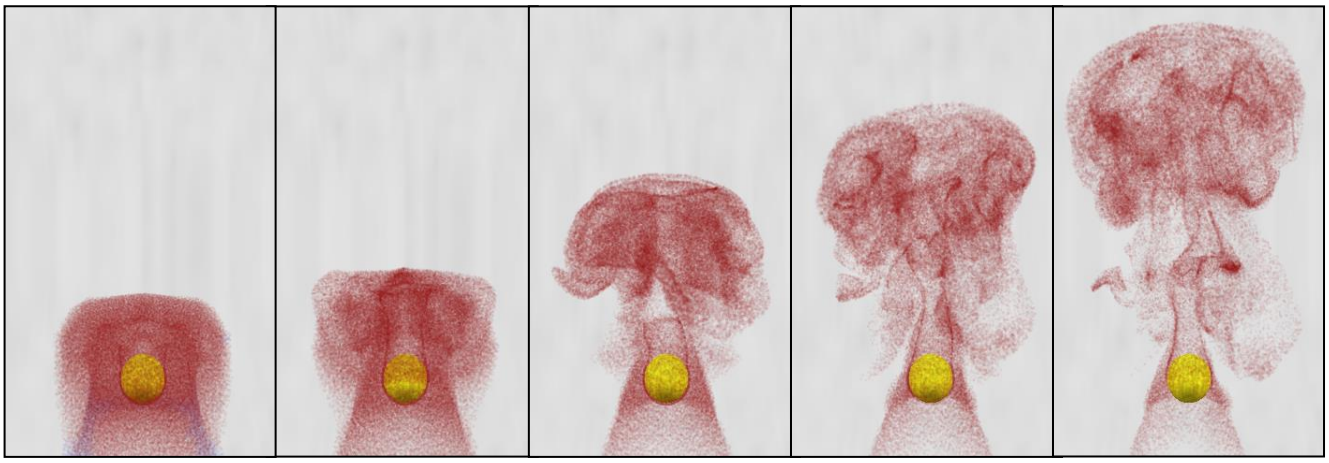

Figure 5: Expanded fluid rises.

Figure 6 shows a rendering of a simulation like that shown in Figures 4 and 5, but with only a thin sheet of dye instead of a thick block. This view more clearly reveals the formation of a pair of counter-rotating vortices that cause the fluid to rise. These vortices formed due to baroclinic generation (as described in part 8).

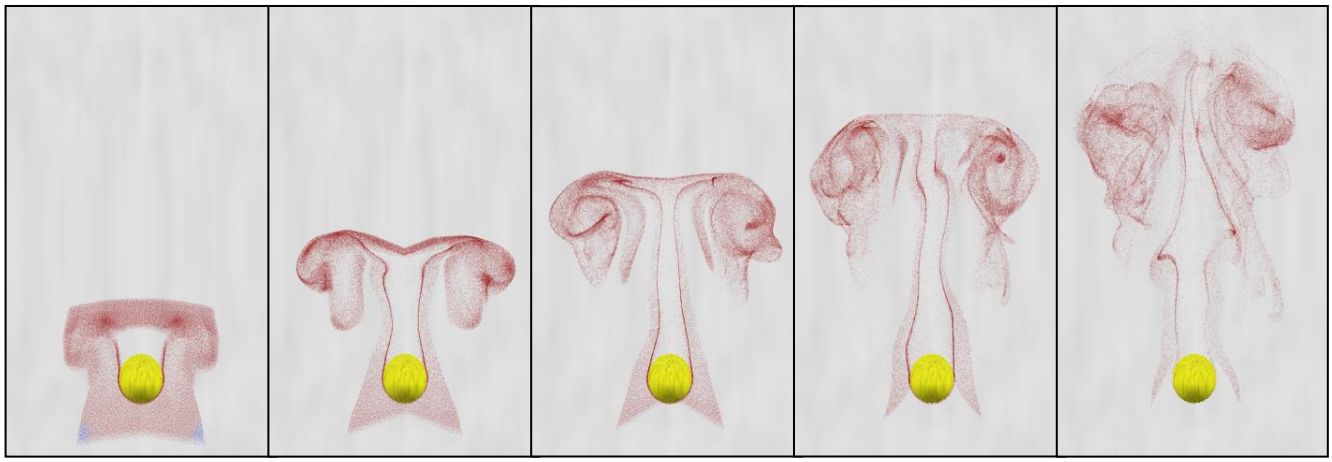

Figure 6: Expanded fluid rises, cross section showing counter-rotating vortices.

Performance

What does all of this additional computation cost? Very little -- less than 1%. Table 1 shows profile measurements.

Note that ParitionVortons was already running before this addition -- it's been there since the first code accompanying any of these articles -- but it used to be incorporated into the DiffuseVortonPSE routine. Now it simply runs in a separate routine. It's a useful benchmark because it is not parallelized, so the runtimes should be roughly independent of the number of threads (and it is).

In contrast, DiffuseHeatPSE uses TBB, and gets significant speed gains as the number of threads increases.

But the total cost of DiffuseHeatPSE is less than 0.2% of the total, regardless of the number of threads.

Table 1. Run Durations for Processes on an Intel® 3.4 GHz Core I7-2600® processor

Threads	DiffuseHeatPSE (ms)	PartitionVortons (ms)	Total (ms)
1	0.021	0.046	10.7
2	0.015	0.048	7.55

4	0.011	0.047	6.46

What did adding thermal expansion cost the simulation? Trick question; it's "free". Or rather, its price was paid back when fluid buoyancy was added and density incorporated into the simulation. Again, no code was added to make "thermal expansion" happen; we simply had the simulation interpret density as both mass-per-volume and an expression of temperature.

Coming Up

The next part will build on this installment, adding *combustion* (generating heat by chemical processes). That addition will give the simulation the ability to simulate smoldering and burning.

Further Reading

Kundo, P.J. (1990): *Fluid Mechanics*. Academic Press, San Diego. See especially chapter 4 (Conservation Laws), section 17(Boussinesq Approximation).

Mandl, F. (1988): Statistics Physics, 2nd Ed.. John Wiley and Sons Ltd., Chichester. See especially Chapter 5 (Simple Thermodynamic Systems) section 3 (The Difference of Heat Capacities) and section 4 (Some Properties of Perfect Gases).