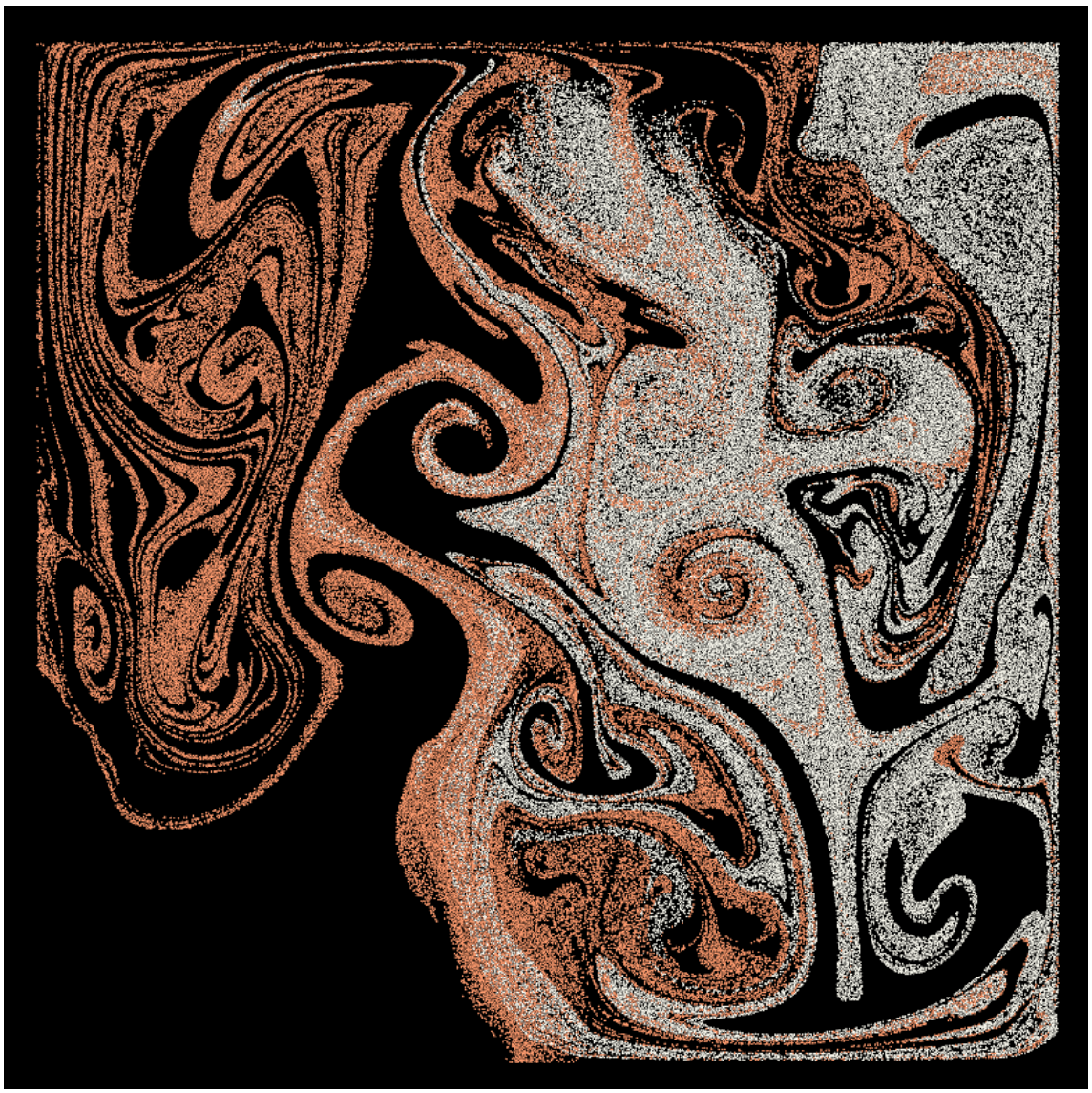


Smoke Simulation

Rishabh Shah, CIS 563 Physically Based Animation 05/06/2018

Introduction

With the increase in computing power, it has now become practical to simulate fluids at large scales with high physical accuracy. These simulations are used everywhere from scientific computing and research, to computer graphics applications. This project is about simulation of smoke.

The simulation uses **Semi-Lagrangian method with MAC-Grid** for simulation of smoke, with **incompressible Euler equations** and a **pressure-Poisson equation**. The model is stable

for any choice of time-step. Smoke has a highly turbulent flow compared to liquid simulations, and thus it suffers from numerical dissipation, which is corrected using a **vorticity confinement term**. The implementation is able to handle boundary conditions in the simulation region, thus enabling addition of obstacles in the path of fluid flow for interesting simulations. The following sections of the readme will cover all of these components in detail.

Marker-and-Cell grid (1965, Hallow)

For Eulerian representation in smoke simulations, MAC grid is the most used method. Here, we store different quantities at different locations. The pressure, density, temperature, etc. are stored at the cell centers, and velocities at cell faces as three separate components *mU, mV, mW*. The velocity at the cell centers can be easily obtained using interpolation.

Advection

For physical simulations, Eulerian or Lagrangian methods can be used. Eulerian method is used for smoke simulations as it is easy to handle topological changes and, collisions and interactions. But it is difficult to follow individual particles in Eulerian view, and so for advection of physical properties, we pull back and solve it like a Lagrangian problem (hence the semi-Lagrangian method).

More specifically, for a quantity q, to find $q^{n+1}(\underline{x}^*)$, we can find an imaginary particle $\underline{X} = \Phi^{-1}(\underline{x}^*, \Delta t)$ at t^n, and grab its Q value. Back tracing is used to find this imaginary particle. In this implementation, 2nd order Runga-Kutta (RK2) is used.

The example code for advection of mU (velocity in X-direction) is shown in Listing 1. Same method can be used for advecting other quantities like density and temperature also.

Listing 1: Sample Advection code – Advecting velocities in X-direction i.e. mU

```
FOR_EACH_FACE {
    // Check if the face is valid
    if (isValidFace(MACGrid::X, i, j, k)) {
        // get current face position
        vec3 currentPt = getFacePosition(MACGrid::X, i, j, k);

        // back trace and get the old position of the imaginary particle
        vec3 oldPt = getRewoundPosition(currentPt, dt);

        // get the velocity at old position
        vec3 newVel = getVelocity(oldPt);
```

```
12
13          // update the current mU
14          target.mU(i, j, k) = newVel[0];
15      }
16      mU = target.mU; // using target as a temporary buffer
17  }
```

External Forces

To make the simulation look realistic, we need to add external environmental forces like gravity and buoyancy. It is done as shown in listing 2. Here, *theBuoyancyAlpha* controls the effect of gravity on the smoke. It is multiplied with density as it represents the mass in unit volume of the grid cell, and is obviously in the downward direction (hence the negative sign). *theBuoyancyBeta* controls the effect of buoyancy due to the temperature difference with the surrounding environment.

Listing 2: Buoyancy and Gravity

```
1   FOR_EACH_FACE{
2       // Check if the face is valid
3       if(isValidFace(MACGrid::Y, i, j, k)) {
4           // get current face position
5           vec3 facePos = getFacePosition(MACGrid::Y, i, j, k);
6
7           // compute the overall effect of buoyancy and gravity
8           double temperatureDifference =
9               getTemperature(facePos) - theBuoyancyAmbientTemperature;
10
11          double buoyancy = - theBuoyancyAlpha * getDensity(facePos)
12                            + theBuoyancyBeta * temperatureDifference;
13
14          // update the Y-velocity as per the buoyancy
15          target.mV(i, j, k) += dt * buoyancy;
16      }
17  }
```

As stated in the introduction, the fluid solver is dissipative and so we need a way to get that energy back in the system. The dissipation here is of rotational energy, and we add that back using *vorticity confinement term* as an external force.

The steps for computing vorticity confinement can be summarized as follows:

1. Compute velocity at cell centers. This can be done simply by averaging velocities at surrounding faces.

```
FOR_EACH_CELL {
    u(i, j, k) = (mU(i + 1, j, k) + mU(i, j, k)) / 2;
    v(i, j, k) = (mV(i, j + 1, k) + mV(i, j, k)) / 2;
    w(i, j, k) = (mW(i, j, k + 1) + mW(i, j, k)) / 2;
}
```

2. Compute vorticities at cell centers using central derivatives.

```
double inv2h = 1.0 / (2.0 * theCellSize);
FOR_EACH_CELL {
    vorticityX(i, j, k) = inv2h * (w(i, j + 1, k) - w(i, j - 1, k) - v(i, j, k + 1) + v(i, j, k - 1));
    vorticityY(i, j, k) = inv2h * (u(i, j, k + 1) - u(i, j, k - 1) - w(i + 1, j, k) + w(i - 1, j, k));
    vorticityZ(i, j, k) = inv2h * (v(i + 1, j, k) - v(i - 1, j, k) - u(i, j + 1, k) + u(i, j - 1, k));
    vorticityLen(i, j, k) = vec3(vorticityX(i, j, k), vorticityY(i, j, k), vorticityZ(i, j, k)).Length();
}
```

3. Compute vorticity gradient and normalize it to get N. Cross product of N and vorticity is used to compute the final force. This is used to update the velocities at cell faces.

```
FOR_EACH_CELL {
    vec3 vortGrad(vorticityLen(i+1, j, k) - vorticityLen(i-1, j, k),
                  vorticityLen(i, j+1, k) - vorticityLen(i, j-1, k),
                  vorticityLen(i, j, k+1) - vorticityLen(i, j, k-1));
    vortGrad *= inv2h;

    vec3 N = vortGrad.Normalize();

    vec3 fconf = theVorticityEpsilon * theCellSize *
                 N.Cross(vec3(vorticityX(i, j, k), vorticityY(i, j, k), vorticityZ(i, j, k)));

    // Apply vortcity confinement to faces..
    // X
    if(isValidFace(MACGrid::X, i, j, k)) {
        target.mU(i, j, k) += dt * fconf[0]/2;
    }
    //// SIMILAR FOR Y AND Z ////

    // X+1
    if(isValidFace(MACGrid::X, i+1, j, k)) {
        target.mU(i+1, j, k) += dt * fconf[0]/2;
    }
    //// SIMILAR FOR Y+1 AND Z+1 ////
}
```

Projection

The projection step in our algorithm is where pressure is computed and updated, and the resultant effect on velocities is reflected in the MAC grid. The steps for computing pressure projection are as follows:

1. Compute the divergence at cell centers. A boundary condition is used here to make sure the cells outside the simulation domain are not used for calculations.

```
FOR_EACH_CELL {
    double velLowX = (i > 0) ? mU(i, j, k) : 0.0;
    double velHighX = (i + 1 < theDim[MACGrid::X]) ? mU(i + 1, j, k) : 0.0;
    double velLowY = (j > 0) ? mV(i, j, k) : 0.0;
    double velHighY = (j + 1 < theDim[MACGrid::Y]) ? mV(i, j + 1, k) : 0.0;
    double velLowZ = (k > 0) ? mW(i, j, k) : 0.0;
    double velHighZ = (k + 1 < theDim[MACGrid::Z]) ? mW(i, j, k + 1) : 0.0;
    d(i, j, k) = -((velHighX - velLowX) + (velHighY - velLowY) + (velHighZ - velLowZ)) / theCellSize;
}
```

2. Compute pressure at cell centers using Preconditioned Conjugate Gradient algorithm

```
preconditionedConjugateGradient(AMatrix, p, d, 200, 0.01);
FOR_EACH_CELL {
    p(i, j, k) *= constMultiplier;
}
```

3. Update velocities using updated pressure.

```
double constMultiplier1 = dt / (theAirDensity * theCellSize);
FOR_EACH_FACE {
    // X
    if (isValidFace(MACGrid::X, i, j, k)) {
        if (isValidCell(i - 1, j, k) && isValidCell(i, j, k)) {
            double deltaPX = p(i, j, k) - p(i - 1, j, k);
            target.mU(i, j, k) -= constMultiplier1 * deltaPX;
        }
        else{
            target.mU(i, j, k) = 0.0;
        }
    }
    //// SIMILAR FOR Y AND Z ////
}
```

Simulation Algorithm

Using all the above concepts, we can summarize each step of the algorithm as follows:

```
1  void SmokeSim::step() {
2      // time step size
3      double dt = 0.1;
4
5      // update the sources - inject smoke into the system
6      mGrid.updateSources();
7
8      // advect velocity
9      mGrid.advectVelocity(dt);
10
11     // add gravity, buoyancy, and vorticity confinement
12     mGrid.addExternalForces(dt);
13
14     // projection step
15     mGrid.project(dt);
16
17     // advect temperature
18     mGrid.advectTemperature(dt);
19
20     // advect density
21     mGrid.advectDensity(dt);
22
23     // advect rendering particles for visualization
24     mGrid.advectRenderingParticles(dt);
```

```
25
26      mTotalFrameNum++;
27  }
```

Multi-threading

As the grid based simulation has a lot of components that can be parallelized, I have added multi-threading to the main simulation code functions such that the simulation can run on 4 threads. Note that only the functions of the simulation step (see previous section) are parallel, the main thread still has to do a lot of extra work in between steps, and for the opengl rendering. So the performance boost is not easily evident. But the simulation step is about 25% faster. This is useful when the opengl preview is not used and only the simulation frames are written out to files. Multithreading only works when grid resolution in Z direction is atleast 4.

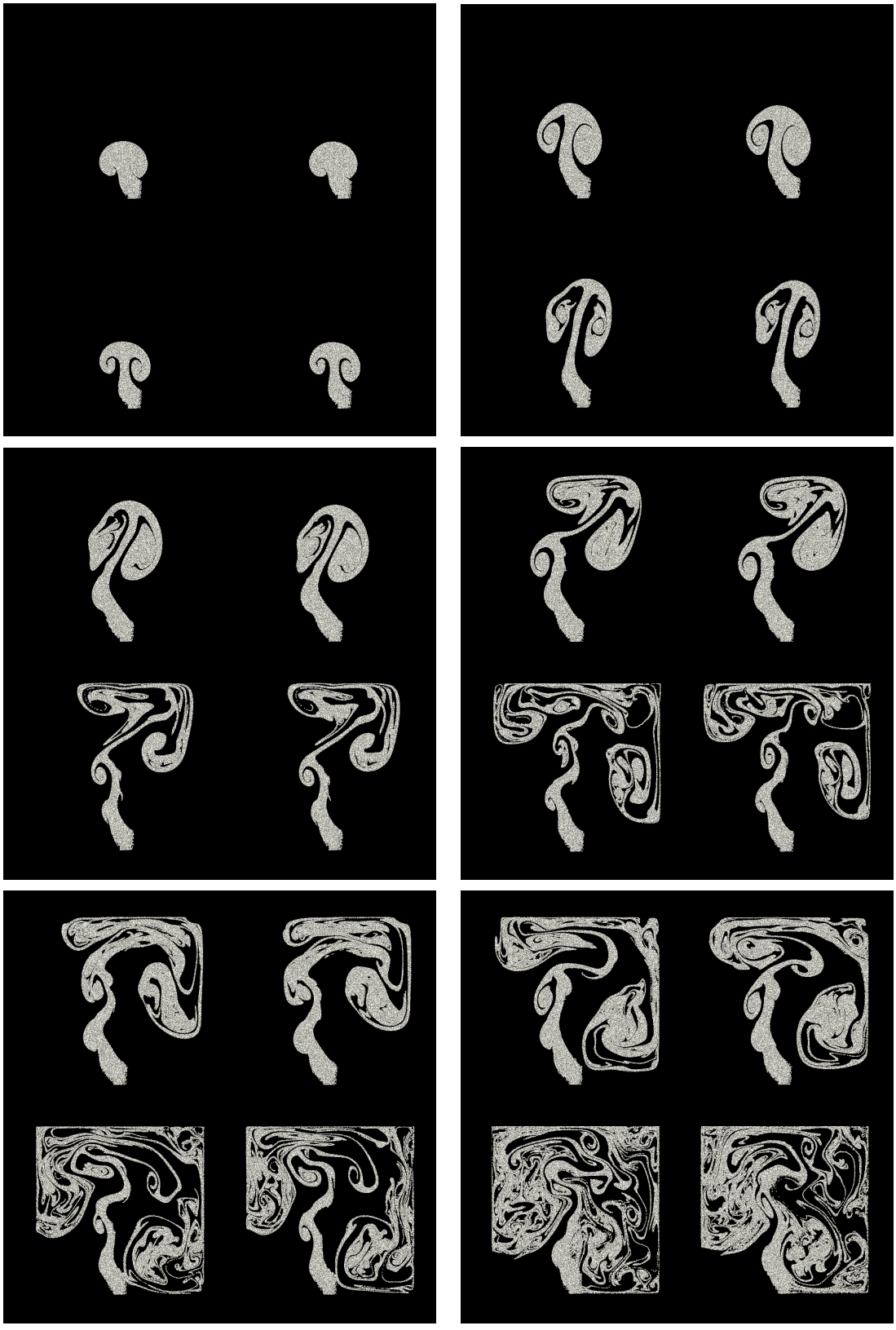
Results

The comparison between smoke with different density and temperature values can be seen in Table 2.

Table 1: Legend for images in Table 2

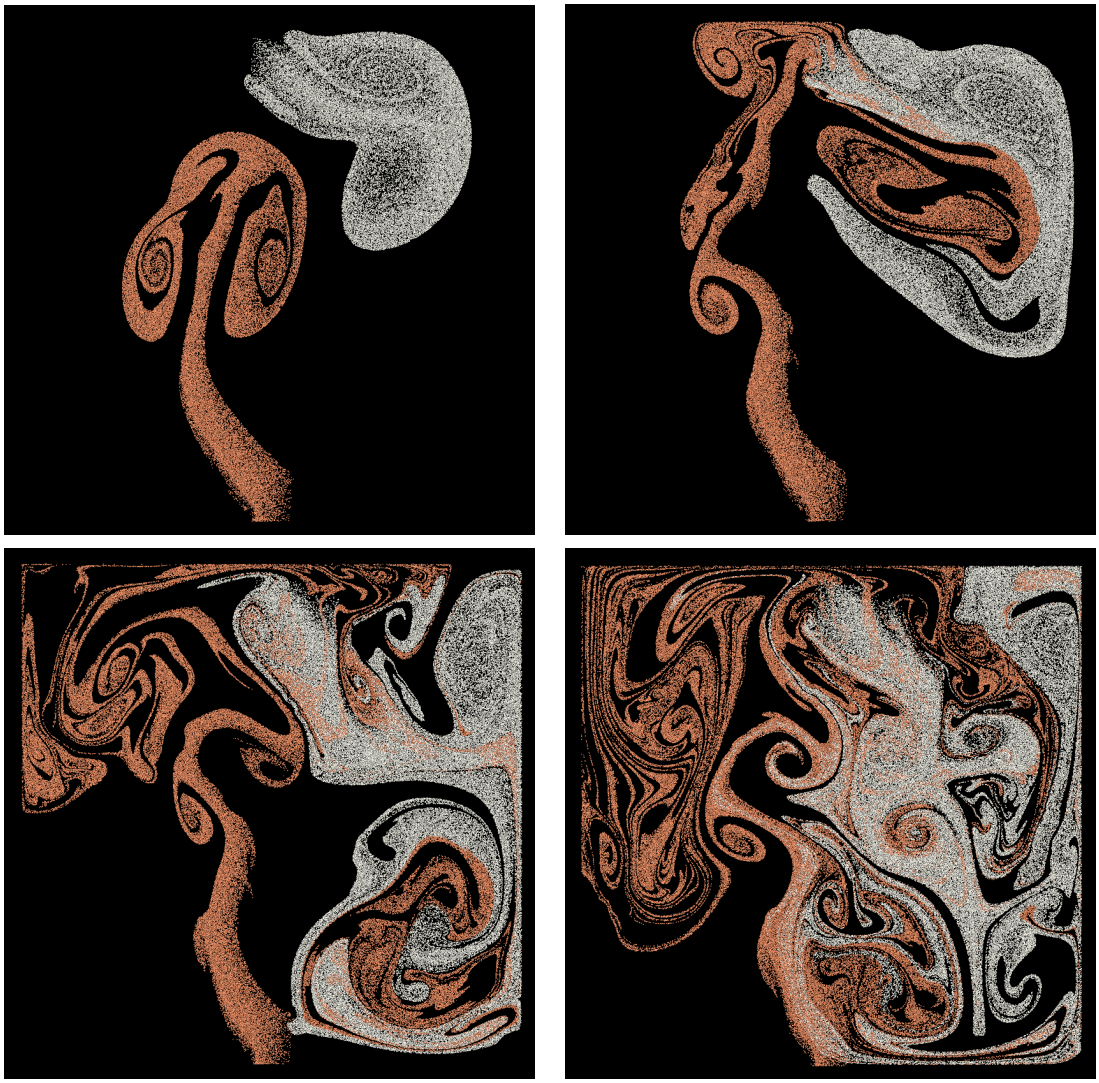
d=0.3, t=1.0	d=0.9, t=1.0
d=0.3, t=3.0	d=0.9, t=3.0

Table 2: Density and temperature comparison



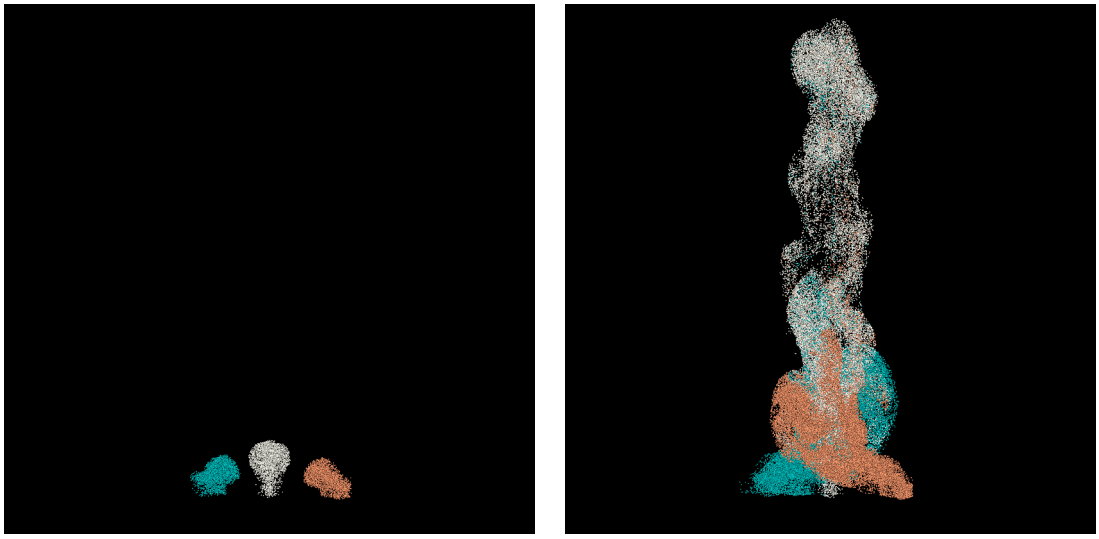
I added ability to give colors to smoke particles for visualization on smoke interaction in Houdini. In Table 3, two smokes of different temperatures can be seen merging.

Table 3: Merging smoke with different temperatures



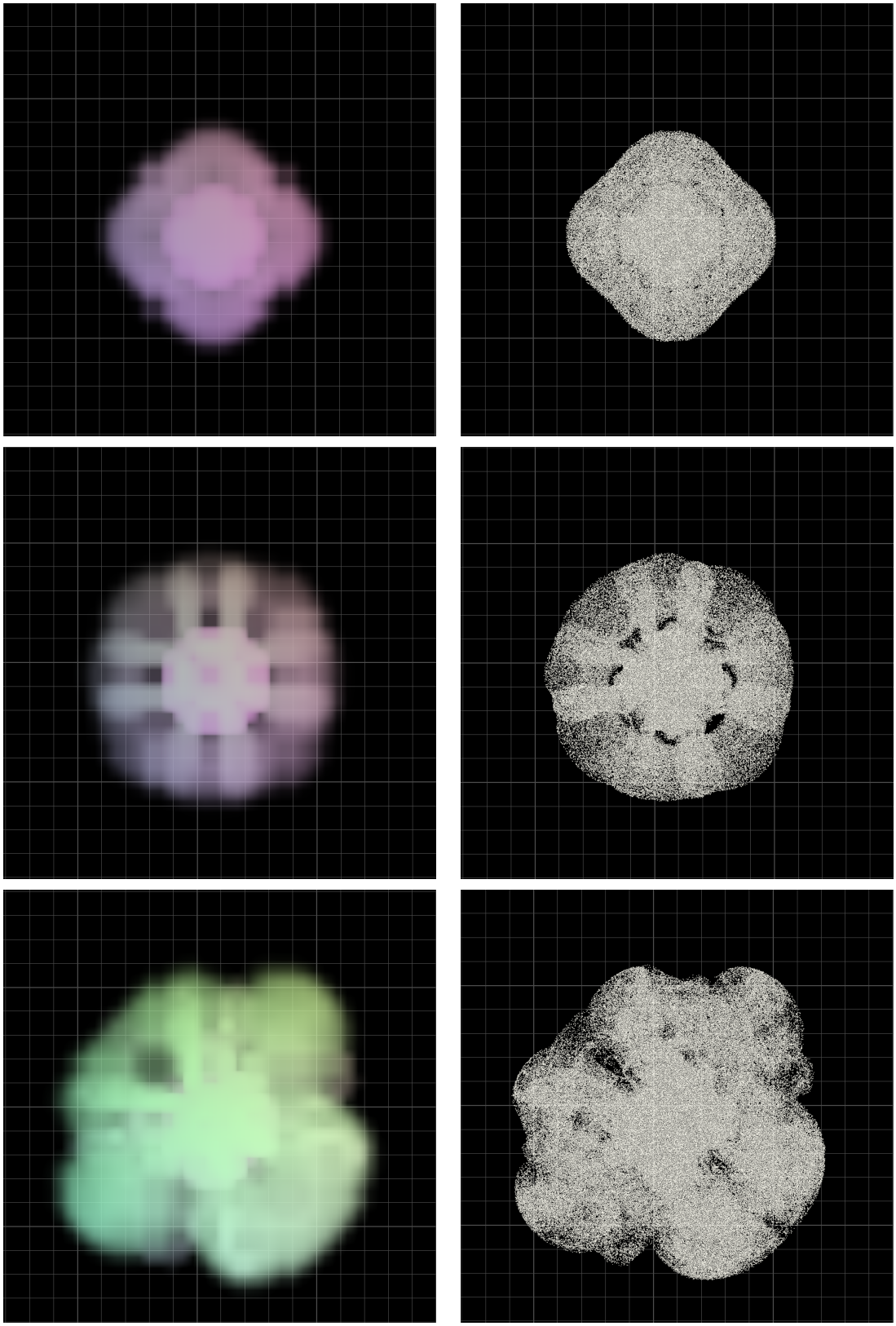
Another example of merging smokes is in Table 4. Here, the sources produce exactly same smoke but with velocities in different directions and with different colors. This simulation is in 3D.

Table 4: Merging smoke



The example in Table 5 shows various frames of smoke interacting with obstacles (like a grid of cubes) placed above the source. The images are from the top view of the 3D simulation. Both density and particle views are shown side by side.

Table 5: Interaction with obstacles - Top view



Videos of these simulations and some other interesting demos can be found along with the submission.

References

1. "Visual Simulation of Smoke", Ronald Fedkiw, Jos Stam, Henrik Wann Jensen
2. "Fluid Simulation SIGGRAPH 2007 Course Notes", Robert Bridson, Matthias Müller-Fischer