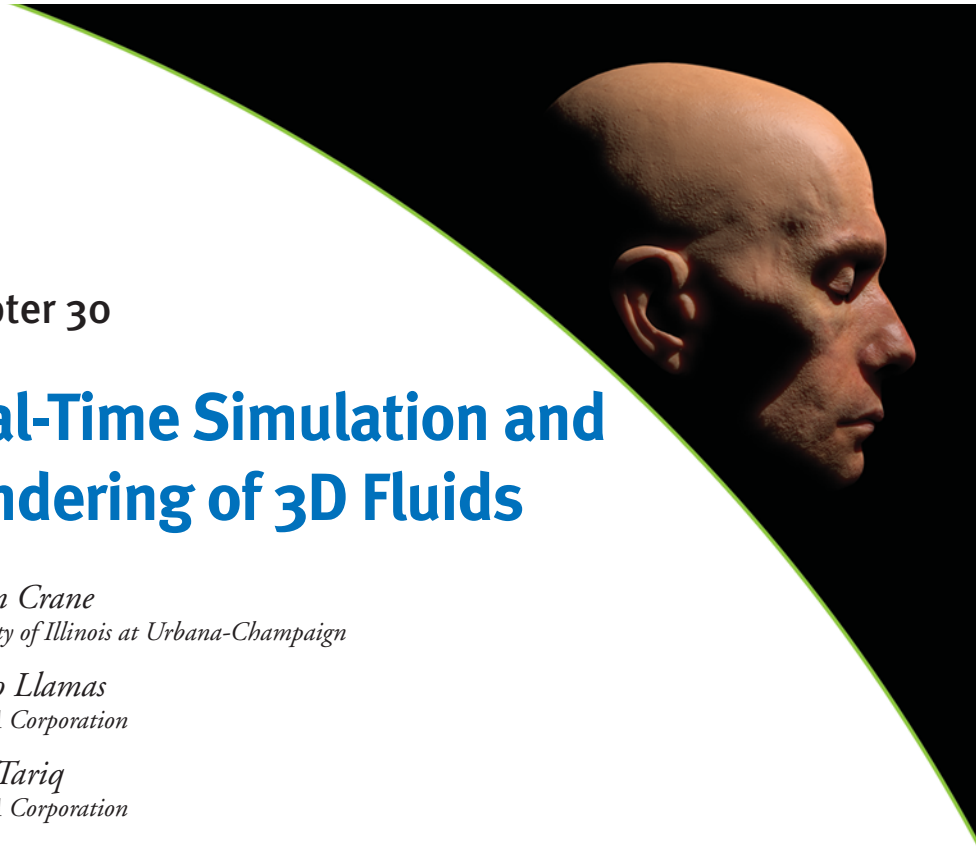

Chapter 30

Real-Time Simulation and Rendering of 3D Fluids

Keenan Crane
University of Illinois at Urbana-Champaign

Ignacio Llamas
NVIDIA Corporation

Sarah Tariq
NVIDIA Corporation

30.1 Introduction

Physically based animation of fluids such as smoke, water, and fire provides some of the most stunning visuals in computer graphics, but it has historically been the domain of high-quality offline rendering due to great computational cost. In this chapter we show not only how these effects can be simulated and rendered in real time, as Figure 30-1 demonstrates, but also how they can be seamlessly integrated into real-time applications. Physically based effects have already changed the way interactive environments are designed. But fluids open the doors to an even larger world of design possibilities.

In the past, artists have relied on particle systems to emulate 3D fluid effects in real-time applications. Although particle systems can produce attractive results, they cannot match the realistic appearance and behavior of fluid simulation. Real time fluids remain a challenge not only because they are more expensive to simulate, but also because the volumetric data produced by simulation does not fit easily into the standard rasterization-based rendering paradigm.

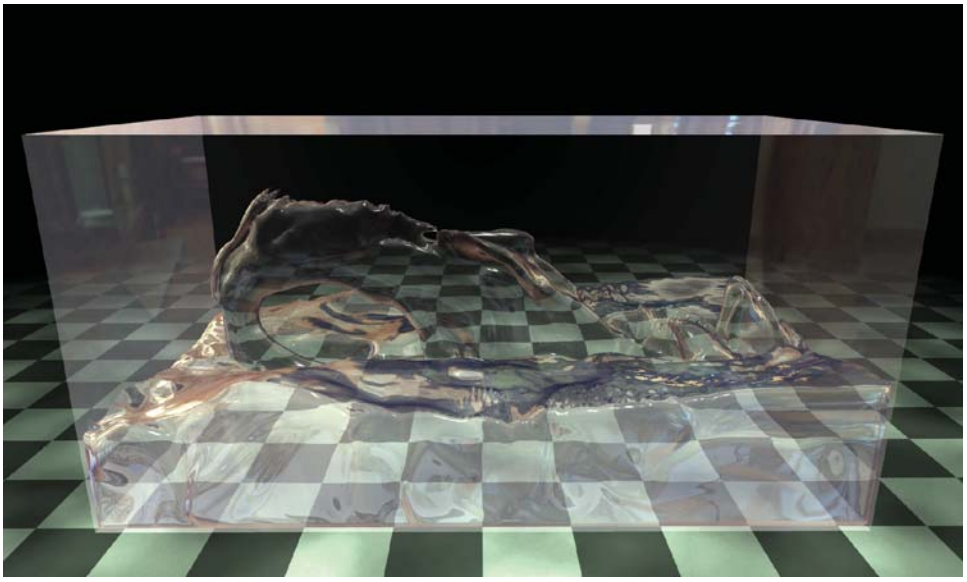

Figure 30-1. Water Simulated and Rendered in Real Time on the GPU

In this chapter we give a detailed description of the technology used for the real-time fluid effects in the NVIDIA GeForce 8 Series launch demo "Smoke in a Box" and discuss its integration into the upcoming game *Hellgate: London*.

The chapter consists of two parts:

- Section 30.2 covers simulation, including smoke, water, fire, and interaction with solid obstacles, as well as performance and memory considerations.
- Section 30.3 discusses how to render fluid phenomena and how to seamlessly integrate fluid rendering into an existing rasterization-based framework.

30.2 Simulation

30.2.1 Background

Throughout this section we assume a working knowledge of general-purpose GPU (GPGPU) methods—that is, applications of the GPU to problems other than conventional raster graphics. In particular, we encourage the reader to look at Harris's chapter on 2D fluid simulation in *GPU Gems* (Harris 2004). As mentioned in that chapter, implementing and debugging a 3D fluid solver is no simple task (even in a traditional programming environment), and a solid understanding of the underlying mathematics

and physics can be of great help. Bridson et al. 2006 provides an excellent resource in this respect.

Fortunately, a deep understanding of partial differential equations (PDEs) is not required to get some basic intuition about the concepts presented in this chapter. All PDEs presented will have the form

$$\frac{\partial}{\partial t}x = f(x,t),$$

which says that the rate at which some quantity x is changing is given by some function f, which may itself depend on x and t. The reader may find it easier to think about this relationship in the discrete setting of *forward Euler integration*:

$$x^{n+1} = x^n + f\left(x^n, t^n\right)\Delta t.$$

In other words, the value of x at the next time step equals the current value of x plus the current rate of change $f(x^n, t^n)$ times the duration of the time step Δt. (Note that superscripts are used to index the time step and do *not* imply exponentiation.) Be warned, however, that the forward Euler scheme is not a good choice numerically—we are suggesting it only as a way to *think* about the equations.

30.2.2 Equations of Fluid Motion

The motion of a fluid is often expressed in terms of its local *velocity* $\mathbf{u}$ as a function of position and time. In computer animation, fluid is commonly modeled as *inviscid* (that is, more like water than oil) and *incompressible* (meaning that volume does not change over time). Given these assumptions, the velocity can be described by the *momentum equation*:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f},$$

subject to the *incompressibility constraint*:

$$\nabla\cdot\mathbf{u} = 0,$$

where p is the pressure, ρ is the mass density, $\mathbf{f}$ represents any external forces (such as gravity), and ∇ is the differential operator:

$$\left[\frac{\partial}{\partial x} \quad \frac{\partial}{\partial y} \quad \frac{\partial}{\partial z}\right]^T.$$

To define the equations of motion in a particular context, it is also necessary to specify *boundary conditions* (that is, how the fluid behaves near solid obstacles or other fluids).

The basic task of a fluid solver is to compute a numerical approximation of **u**. This velocity field can then be used to animate visual phenomena such as smoke particles or a liquid surface.

30.2.3 Solving for Velocity

The popular "stable fluids" method for computing velocity was introduced in Stam 1999, and a GPU implementation of this method for 2D fluids was presented in Harris 2004. In this section we briefly describe how to solve for velocity but refer the reader to the cited works for details.

In order to numerically solve the momentum equation, we must *discretize* our domain (that is, the region of space through which the fluid flows) into computational elements. We choose an *Eulerian* discretization, meaning that computational elements are fixed in space throughout the simulation—only the values stored on these elements change. In particular, we subdivide a rectilinear volume into a regular grid of cubical cells. Each grid cell stores both scalar quantities (such as pressure, temperature, and so on) and vector quantities (such as velocity). This scheme makes implementation on the GPU simple, because there is a straightforward mapping between grid cells and voxels in a 3D texture. *Lagrangian* schemes (that is, schemes where the computational elements are *not* fixed in space) such as smoothed-particle hydrodynamics (Müller et al. 2003) are also popular for fluid animation, but their irregular structure makes them difficult to implement efficiently on the GPU.

Because we discretize space, we must also discretize *derivatives* in our equations: *finite differences* numerically approximate derivatives by taking linear combinations of values defined on the grid. As in Harris 2004, we store all quantities at cell centers for pedagogical simplicity, though a staggered MAC-style grid yields more-robust finite differences and can make it easier to define boundary conditions. (See Harlow and Welch 1965 for details.)

In a GPU implementation, cell attributes (velocity, pressure, and so on) are stored in several 3D textures. At each simulation step, we update these values by running computational *kernels* over the grid. A kernel is implemented as a pixel shader that executes on every cell in the grid and writes the results to an output texture. However, because

GPUs are designed to render into 2D buffers, we must run kernels once for each slice of a 3D volume.

To execute a kernel on a particular grid slice, we rasterize a single quad whose dimensions equal the width and height of the volume. In Direct3D 10 we can directly render into a 3D texture by specifying one of its slices as a render target. Placing the slice index in a variable bound to the `SV_RenderTargetArrayIndex` semantic specifies the slice to which a primitive coming out of the geometry shader is rasterized. (See Blythe 2006 for details.) By iterating over slice indices, we can execute a kernel over the entire grid.

Rather than solve the momentum equation all at once, we split it into a set of simpler operations that can be computed in succession: advection, application of external forces, and pressure projection. Implementation of the corresponding kernels is detailed in Harris 2004, but several examples from our Direct3D 10 framework are given in Listing 30-1. Of particular interest is the routine `PS_ADVECT_VEL`: this kernel implements *semi-Lagrangian* advection, which is used as a building block for more accurate advection in the next section.

Listing 30-1. Simulation Kernels

```
struct GS_OUTPUT_FLUIDSIM
{
  // Index of the current grid cell (i,j,k in [0,gridSize] range)
  float3 cellIndex : TEXCOORD0;

  // Texture coordinates (x,y,z in [0,1] range) for the
  // current grid cell and its immediate neighbors
  float3 CENTERCELL : TEXCOORD1;
  float3 LEFTCELL   : TEXCOORD2;
  float3 RIGHTCELL  : TEXCOORD3;
  float3 BOTTOMCELL : TEXCOORD4;
  float3 TOPCELL    : TEXCOORD5;
  float3 DOWNCELL   : TEXCOORD6;
  float3 UPCELL     : TEXCOORD7;
  float4 pos        : SV_Position; // 2D slice vertex in
                                   // homogeneous clip space
  uint RTIndex     : SV_RenderTargetArrayIndex; // Specifies
                                                // destination slice
};
```

Listing 30-1 (*continued*). Simulation Kernels

```
float3 cellIndex2TexCoord(float3 index)
{
  // Convert a value in the range [0,gridSize] to one in the range [0,1].
  return float3(index.x / textureWidth,
                index.y / textureHeight,
                (index.z+0.5) / textureDepth);
}

float4 PS_ADVECT_VEL(GS_OUTPUT_FLUIDSIM in,
                     Texture3D velocity) : SV_Target
{
  float3 pos = in.cellIndex;
  float3 cellVelocity = velocity.Sample(samPointClamp,
                                        in.CENTERCELL).xyz;

  pos -= timeStep * cellVelocity;
  pos = cellIndex2TexCoord(pos);

  return velocity.Sample(samLinear, pos);
}

float PS_DIVERGENCE(GS_OUTPUT_FLUIDSIM in,
                    Texture3D velocity) : SV_Target
{
  // Get velocity values from neighboring cells.
  float4 fieldL = velocity.Sample(samPointClamp, in.LEFTCELL);
  float4 fieldR = velocity.Sample(samPointClamp, in.RIGHTCELL);
  float4 fieldB = velocity.Sample(samPointClamp, in.BOTTOMCELL);
  float4 fieldT = velocity.Sample(samPointClamp, in.TOPCELL);
  float4 fieldD = velocity.Sample(samPointClamp, in.DOWNCELL);
  float4 fieldU = velocity.Sample(samPointClamp, in.UPCELL);

  // Compute the velocity's divergence using central differences.
  float divergence =  0.5 * ((fieldR.x - fieldL.x)+
                             (fieldT.y - fieldB.y)+
                             (fieldU.z - fieldD.z));

  return divergence;
}
```

Listing 30-1 (*continued*). Simulation Kernels

```
float PS_JACOBI(GS_OUTPUT_FLUIDSIM in,
                Texture3D pressure,
                Texture3D divergence) : SV_Target
{
  // Get the divergence at the current cell.
  float dC = divergence.Sample(samPointClamp, in.CENTERCELL);

  // Get pressure values from neighboring cells.
  float pL = pressure.Sample(samPointClamp, in.LEFTCELL);
  float pR = pressure.Sample(samPointClamp, in.RIGHTCELL);
  float pB = pressure.Sample(samPointClamp, in.BOTTOMCELL);
  float pT = pressure.Sample(samPointClamp, in.TOPCELL);
  float pD = pressure.Sample(samPointClamp, in.DOWNCELL);
  float pU = pressure.Sample(samPointClamp, in.UPCELL);

  // Compute the new pressure value for the center cell.
  return(pL + pR + pB + pT + pU + pD - dC) / 6.0;
}

float4 PS_PROJECT(GS_OUTPUT_FLUIDSIM in,
                  Texture3D pressure,
                  Texture3D velocity): SV_Target
{
  // Compute the gradient of pressure at the current cell by
  // taking central differences of neighboring pressure values.
  float pL = pressure.Sample(samPointClamp, in.LEFTCELL);
  float pR = pressure.Sample(samPointClamp, in.RIGHTCELL);
  float pB = pressure.Sample(samPointClamp, in.BOTTOMCELL);
  float pT = pressure.Sample(samPointClamp, in.TOPCELL);
  float pD = pressure.Sample(samPointClamp, in.DOWNCELL);
  float pU = pressure.Sample(samPointClamp, in.UPCELL);
  float3 gradP = 0.5*float3(pR - pL, pT - pB, pU - pD);

  // Project the velocity onto its divergence-free component by
  // subtracting the gradient of pressure.
  float3 vOld = velocity.Sample(samPointClamp, in.texcoords);
  float3 vNew = vOld - gradP;

  return float4(vNew, 0);
}
```

Improving Detail

The semi-Lagrangian advection scheme used by Stam is useful for animation because it is unconditionally stable, meaning that large time steps will not cause the simulation to "blow up." However, it can introduce unwanted numerical smoothing, making water look viscous or causing smoke to lose detail. To achieve higher-order accuracy, we use a MacCormack scheme that performs two intermediate semi-Lagrangian advection steps. Given a quantity ϕ and an advection scheme A (for example, the one implemented by `PS_ADVECT_VEL`), higher-order accuracy is obtained using the following sequence of operations (from Selle et al. 2007):

$$\hat{\phi}^{n+1} = A\left(\phi^n\right)$$

$$\hat{\phi}^{n} = A^R\left(\hat{\phi}^{n+1}\right)$$

$$\phi^{n+1} = \hat{\phi}^{n+1} + \frac{1}{2}\left(\phi^n - \hat{\phi}^n\right).$$

Here, ϕ^n is the quantity to be advected, $\hat{\phi}^{n+1}$ and $\hat{\phi}^n$ are intermediate quantities, and ϕ^{n+1} is the final advected quantity. The superscript on A^R indicates that advection is reversed (that is, time is run backward) for that step.

Unlike the standard semi-Lagrangian scheme, this MacCormack scheme is not unconditionally stable. Therefore, a limiter is applied to the resulting value ϕ^{n+1}, ensuring that it falls within the range of values contributing to the initial semi-Lagrangian advection. In our GPU solver, this means we must locate the eight nodes closest to the sample point, access the corresponding texels *exactly at their centers* (to avoid getting interpolated values), and clamp the final value to fall within the minimum and maximum values found on these nodes, as shown in Figure 30-2.

Once the intermediate semi-Lagrangian steps have been computed, the pixel shader in Listing 30-2 completes advection using the MacCormack scheme.

Listing 30-2. MacCormack Advection Scheme

```
float4 PS_ADVECT_MACCORMACK(GS_OUTPUT_FLUIDSIM in,
                            float timestep) : SV_Target
{
  // Trace back along the initial characteristic - we'll use
  // values near this semi-Lagrangian "particle" to clamp our
  // final advected value.
  float3 cellVelocity = velocity.Sample(samPointClamp,
                                        in.CENTERCELL).xyz;
```

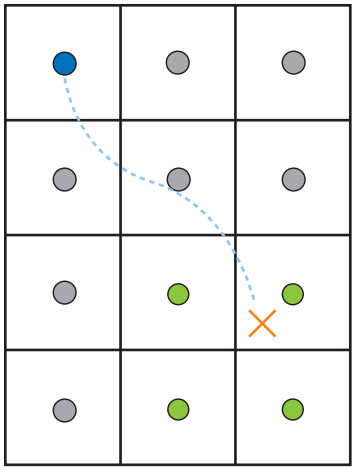

Figure 30-2. Limiter Applied to a MacCormack Advection Scheme in 2D
The result of the advection (blue) is clamped to the range of values from nodes (green) used to get the interpolated value at the advected "particle" (red) in the initial semi-Lagrangian step.

Listing 30-2 (*continued*). MacCormack Advection Scheme

```
float3 npos = in.cellIndex - timestep * cellVelocity;

// Find the cell corner closest to the "particle" and compute the
// texture coordinate corresponding to that location.
npos = floor(npos + float3(0.5f, 0.5f, 0.5f));
npos = cellIndex2TexCoord(npos);

// Get the values of nodes that contribute to the interpolated value.

// Texel centers will be a half-texel away from the cell corner.
float3 ht = float3(0.5f / textureWidth,
                   0.5f / textureHeight,
                   0.5f / textureDepth);

float4 nodeValues[8];
nodeValues[0] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x, -ht.y, -ht.z));
nodeValues[1] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x, -ht.y,  ht.z));
nodeValues[2] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x,  ht.y, -ht.z));
nodeValues[3] = phi_n.Sample(samPointClamp, npos +
                             float3(-ht.x,  ht.y,  ht.z));
```

Listing 30-2 (*continued*). MacCormack Advection Scheme

```
    nodeValues[4] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x, -ht.y, -ht.z));
    nodeValues[5] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x, -ht.y,  ht.z));
    nodeValues[6] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x,  ht.y, -ht.z));
    nodeValues[7] = phi_n.Sample(samPointClamp, npos +
                                 float3(ht.x,  ht.y,  ht.z));

    // Determine a valid range for the result.
    float4 phiMin = min(min(min(min(min(min(min(
        nodeValues[0],  nodeValues [1]), nodeValues [2]), nodeValues [3]),
        nodeValues[4]), nodeValues [5]), nodeValues [6]), nodeValues [7]);

    float4 phiMax = max(max(max(max(max(max(max(
        nodeValues[0],  nodeValues [1]), nodeValues [2]), nodeValues [3]),
        nodeValues[4]), nodeValues [5]), nodeValues [6]), nodeValues [7]);

    // Perform final advection, combining values from intermediate
    // advection steps.
    float4 r = phi_n_1_hat.Sample(samLinear, nposTC) +
               0.5 * (phi_n.Sample(samPointClamp, in.CENTERCELL) -
                      phi_n_hat.Sample(samPointClamp, in.CENTERCELL));

    // Clamp result to the desired range.
    r = max(min(r, phiMax), phiMin);

    return r;
}
```

On the GPU, higher-order schemes are often a better way to get improved visual detail than simply increasing the grid resolution, because math is cheap compared to bandwidth. Figure 30-3 compares a higher-order scheme on a low-resolution grid with a lower-order scheme on a high-resolution grid.

30.2.4 Solid-Fluid Interaction

One of the benefits of using real-time simulation (versus precomputed animation) is that fluid can interact with the environment. Figure 30-4 shows an example on one such scene. In this section we discuss two simple ways to allow the environment to act on the fluid.

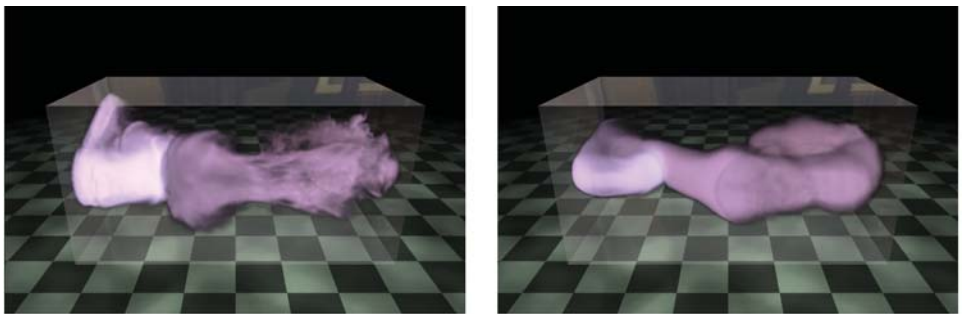

Figure 30-3. Bigger Is Not Always Better!
Left: MacCormack advection scheme (applied to both velocity and smoke density) on a 128×64×64 grid. Right: Semi-Lagrangian advection scheme on a 256×128×128 grid.

A basic way to influence the velocity field is through the application of external forces. To get the gross effect of an obstacle pushing fluid around, we can approximate the obstacle with a basic shape such as a box or a ball and add the obstacle's average velocity to that region of the velocity field. Simple shapes like these can be described with an implicit equation of the form $f(x, y, z) \leq 0$ that can be easily evaluated by a pixel shader at each grid cell.

Although we could explicitly add velocity to approximate simple motion, there are situations in which more detail is required. In *Hellgate: London*, for example, we wanted smoke to seep out through cracks in the ground. Adding a simple upward velocity and smoke density in the shape of a crack resulted in uninteresting motion. Instead, we used the crack shape, shown inset in Figure 30-5, to define *solid obstacles* for smoke to collide and interact with. Similarly, we wanted to achieve more-precise interactions between smoke and an animated gargoyle, as shown in Figure 30-4. To do so, we needed to be able to affect the fluid motion with dynamic obstacles (see the details later in this section), which required a volumetric representation of the obstacle's interior and of the velocity at its boundary (which we also explain later in this section).

Figure 30-4. An Animated Gargoyle Pushes Smoke Around by Flapping Its Wings