Gases and Fluids

Simple, Fast, and Controllable Gas and Fluid Effects for Computer Games

Ben Kenwright

Abstract—Generating life-like virtual smoke and liquid effects in real-time that are interactive and dynamic produces more immersive, engaging, and addictive virtual worlds; for example, computer games with sea effects and city fog that realistically flows and moves around buildings and vehicles. However, creating smoke and fluid effects that can run in real-time, such as in games, in a realistic, physically-correct, and life-like manner is challenging, interesting, and important. The challenges stem from the fact that synthesizing smoke and fluid in a natural manner is complex due to the numerical complexity of the problem (e.g., conservation of energy, density, and the Navier-Stokes equations). One fundamental effect that is crucial for realistic smoke and fluid is the dynamic dispersion and swirling. Furthermore, in interactive environments, the smoke and liquid need to respond to disturbances realistically, i.e., automatically react by moving in a way we would expect in the real world. This paper explains the fundamental principles and simple numerical tips and tricks in a step-by-step manner for creating realistic real-time smoke and fluid effects. Furthermore, we include practical source code implementations and numerical details. We demonstrate how to avoid common visual artifacts and physically-implausible results (e.g., the instability and unnatural mixing of liquids and gases).

Index Terms—Smoke, Real-Time, Controllable, Interactive, Gas, Navier-Stokes, Solver, Fluid Mechanics, Fluid, Real-Time, Computer Games, Interactive

Fig. 1. Smoke Effect. Screen capture of the simulation showing the mouse cursor drawing onto a blank canvas and having it disperse in a smoke like effect (i.e., also analogous to dropping dye into water).

1 Introduction

Fluid dynamics are a common site in interactive virtual worlds, such as video-games and training simulations. However, creating 'fast' gas and fluid simulations that 'appear' physically accurate, controllable, and interactive on-the-fly and in real-time that mimic the real-world is highly challenging, interesting, and important. This is because fluids and gases can bring an otherwise static and uninteresting scene to life. For example, clouds, rivers, and steam. There is a vast assortment of diverse, original, and complex techniques that are both physically bound (i.e., numerically accurate); however, this paper focuses on less accurate techniques that are 'visually' life-like and ideal for interactive environments, such as video games.

1.1 Motivation

The motivation for this paper...

1.2 Contribution

The key contributions of this paper are..

• *Ben Kenwright*
E-mail: bkenwright@xbdev.net

Crucial Features and Points of Interest:
o Constant energy
o Stability
o Real-Time
o Uncomplicated
o Memory/Bandwidth efficient
o Realistic looking

o Different gases mix
o 2D and 3D o 2 1/2D (Mesh surface raised)

2 Particles or Grids

Two main approaches for generating gas like effects are the particle based technique and the grid based technique.

The particle approach, the scene is filled with lots of particles that move around and interactive with one another.

The grid-based approach, subdivides the scene into a grid with each grid cell containing a numerical value. Depending upon the technique the cell sizes can be dynamic and diverse including how the values interact and change between frame updates.

3 Naive Approach

An extremely simple and intuitive first approach is the "blur" based technique. Essentially, each frame, the cell pixel

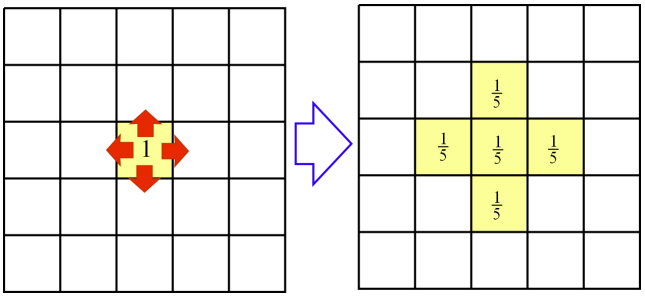

Fig. 2. **Gas Diffusion** - Iteratively merging cells between themselves and their neighbors causes a dispersive type effect

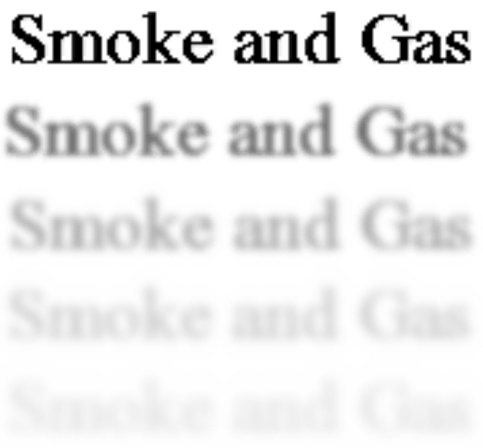

Fig. 3. **Blending Screenshot** - Step-by-step screenshot of the image fading away if we blend surrounding cells with a (300x200) grid (top to bottom increasing with time).

value is subdivided between itself and its neighboring cells.

```
void BlendUpdate()
{
  float[,] tempCellData = new float[m_gridSize.X,m_gridSize.Y];
  for (int x = 0; x < m_gridSize.X-1; x++)
  {
  for (int y = 0; y < m_gridSize.Y-1; y++)
  {
    float middle = m_cellData[x,y];
    // We can't take the modulus of a negative number so we
    // wrap it around by adding the width/height
    float top    = m_cellData[x,(m_gridSize.Y+y-1)%m_gridSize.Y];
    float bottom = m_cellData[x,(m_gridSize.Y+y+1)%m_gridSize.Y];
    float left   = m_cellData[(m_gridSize.X+x-1)%m_gridSize.X,y];
    float right  = m_cellData[(m_gridSize.X+x+1)%m_gridSize.X,y];
    tempCellData[x,y] += (middle+top+bottom+left+right)/6.0f;
  }
  }
  m_cellData = tempCellData;
}
```

3.1 Is it stable?

While you might think it can't be unstable, in retrospect it can. If we assume the overall density of the grid should remain constant, small numerical errors can creep in over time and eventually cause the simulation to produce erratic results.

For example, dividing the five cells by 5, keeps the density of about approx 1029 (after 2 minutes the density is 1028.89). We can divide by 6 to make the result fade away quicker and have the density diminish towards zero quickly. However, if we use a value only slightly less than 5 (e.g., 4.9), the density will increase quickly going to infinity and ultimately crashing.

3.1.1 How can we force it stable?

We can force the simulation to remain more stable - even when we use a value of 4.0 for the divisor, so the overall density increases over time. We calculate the total density (i.e., the sum of all the cells) at the start of the blur, the compare it will what it's after the blur. We then 'scale' all the cells by the error so that the density remains constant.

Note, you can't just subtract the error across all the cells, since some cells might be zero, and you'll end up with negative density. Furthermore, if you clamp the negative values, it will only lead to more sporadic jumps since the average total density won't remain constant.

3.2 What's Missing

The uncomplicated blur approach is effective and fast it has a number of drawbacks.

- no velocity information
- no twist and swirl artifacts

4 Simple Optimization

Some simple tips to achieve a faster simulation:

- Only draw the cells that have something in (e.g., cell > 0)

5 Summary

We are going to see film and game animations take on a new form. Similar, to how we saw computer generated graphics replace traditional hand drawn scenes, we will see procedural physics-based solutions replace traditional pre-recorded key-framed motion capture solutions. Exploiting techniques from multiple research disciplines (such as biomechanics, robotics, and computer science) to create intelligent self driven characters.

Acknowledgments

A special thanks to readers for taking the time to contact me and provide insightful, invaluable comments and suggestions to help to improve the quality of this article.

Appendix

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using System.Windows.Forms;


class DoubleArray
{
  float[,]  m_val = null;
  int       m_x   = 0;
  int       m_y   = 0;

  public DoubleArray(int nx, int ny)
  {
    m_x = nx;
    m_y = ny;
    m_val = new float[nx, ny];
```

Fig. 4. **Sample Text Smoke Effect Screenshot** - Step-by-step screenshot of the sample text fading away (220x220) grid.

```
  }

  public void Clear()
  {
    for (int i=0; i<m_x; ++i)
    { for (int j=0; j<m_y; ++j)
      {
        m_val[i,j] = 0;
      }
    }
  }

  public float this[int i, int j]
  {
    get
    {
      if ( i<0 )     i = 0;
      if ( i>m_x-1) i = m_x-1;
      if ( j<0 )     j = 0;
      if ( j>m_y-1) j = m_y-1;
      return m_val[i,j];
    }

    set
    {
      if ( i<0 )     i = 0;
      if ( i>m_x-1) i = m_x-1;
      if ( j<0 )     j = 0;
      if ( j>m_y-1) j = m_y-1;
      m_val[i,j] = value;
    }
  }
}

class SmokeSolver
{
  public int    n;
  //public int    size;
  public float  dt;

  public float  visc = 0.0f;
  public float  diff = 0.0f;

  public DoubleArray  curl, tmp;
  public DoubleArray  d,    dOld;
  public DoubleArray  u,    uOld;
  public DoubleArray  v,    vOld;


  // Helper swapping methods
  public void SwapU(){ tmp = u; u = uOld; uOld = tmp; }
  public void SwapV(){ tmp = v; v = vOld; vOld = tmp; }
  public void SwapD(){ tmp = d; d = dOld; dOld = tmp; }

  public void Setup(int n, float dt)
  {
    this.n  = n;
    this.dt = dt;
    Reset();
  }

  public void Reset()
  {
    d    = new DoubleArray(n,n);
    dOld = new DoubleArray(n,n);
    u    = new DoubleArray(n,n);
    uOld = new DoubleArray(n,n);
    v    = new DoubleArray(n,n);
    vOld = new DoubleArray(n,n);
    curl = new DoubleArray(n,n);
  }


  public void Buoyancy(DoubleArray Fbuoy)
  {
    float Tamb  = 0;
    float a   = 0.000625f;
    float b   = 0.025f;

    // sum all temperatures
    for (int i = 0; i<n; ++i)
    { for (int j = 0; j<n; ++j)
      {
        Tamb += d[i,j];
      }
    }

    // get average temperature
    Tamb /= (n * n);

    // for each cell compute buoyancy force
    for (int i = 0; i<n; ++i)
    { for (int j = 0; j<n; ++j)
      {
        Fbuoy[i, j] = a * d[i,j] + -b * (d[i,j] - Tamb);
      }
    }
  }

  public float Curl(int i, int j)
  {
    float du_dy = (u[i, j + 1] - u[i, j - 1]) * 0.5f;
    float dv_dx = (v[i + 1, j] - v[i - 1, j]) * 0.5f;
    return du_dy - dv_dx;
  }

  public void VorticityConfinement(DoubleArray Fvc_x, DoubleArray Fvc_y)
  {
    float dw_dx, dw_dy;
    float length;
    float v;

    // Calculate magnitude of curl(u,v) for each cell. (|w|)
    for (int i = 0; i<n; ++i)
    { for (int j = 0; j<n; ++j)
      {
        curl[i,j] = (float)Math.Abs(Curl(i, j));
      }
    }

    for (int i=0; i<n; ++i)
    { for (int j=0; j<n; ++j)
      {
        // Find derivative of the magnitude (n = del |w|)
        dw_dx = (curl[i + 1, j] - curl[i - 1, j]) * 0.5f;
```

```
        dw_dy = (curl[i, j + 1] - curl[i, j - 1]) * 0.5f;

        // Calculate vector length. (|n|)
        // Add small factor to prevent divide by zeros.
        length = (float) Math.Sqrt(dw_dx * dw_dx + dw_dy * dw_dy)
                 + 0.000001f;

        // N = ( n/|n| )
        dw_dx /= length;
        dw_dy /= length;

        v = Curl(i, j);

        // N x w
        Fvc_x[i, j] = dw_dy * -v;
        Fvc_y[i, j] = dw_dx *  v;
      }
    }
  }

  public void VelocitySolver()
  {

    // add velocity that was input by mouse
    AddSource(u, uOld);
    AddSource(v, vOld);

    // add in vorticity confinement force
    VorticityConfinement(uOld, vOld);
    AddSource(u, uOld);
    AddSource(v, vOld);

    // add in buoyancy force
    Buoyancy(vOld);
    AddSource(v, vOld);

    // swapping arrays for economical mem use
    // and calculating diffusion in velocity.
    SwapU();
    Diffuse(0, u, uOld, visc);

    SwapV();
    Diffuse(0, v, vOld, visc);

    // we create an incompressible field
    // for more effective advection.
    Project(u, v, uOld, vOld);

    SwapU();
    SwapV();

    // self advect velocities
    Advect(1, u, uOld, uOld, vOld);
    Advect(2, v, vOld, uOld, vOld);

    // make an incompressible field
    Project(u, v, uOld, vOld);

    // clear all input velocities for next frame
    uOld.Clear();
    vOld.Clear();
  }

  public void DensitySolver()
  {
    // Add density inputted by mouse
    AddSource(d, dOld);
    SwapD();

    Diffuse(0, d, dOld, diff);
    SwapD();

    Advect(0, d, dOld, u, v);

    // Clear input density array for next frame
    dOld.Clear();
  }

  private void AddSource(DoubleArray x, DoubleArray x0)
  {
    for (int i=0; i<n; i++)
    { for (int j=0; j<n; ++j)
      {
        x[i,j] += dt * x0[i,j];
      }
    }
  }

  private void Advect(int b, DoubleArray d, DoubleArray d0, DoubleArray du, DoubleArray dv)
  {
    float dt0 = dt * n;

    for (int i = 0; i < n; ++i)
    { for (int j = 0; j < n; ++j)
      {
        // Go backwards through velocity field
        float x = i - dt0 * du[i, j];
        float y = j - dt0 * dv[i, j];

        // Interpolate results
        if (x > n + 0.5) x = n + 0.5f;
        if (x < 0.5)     x = 0.5f;

        int i0 = (int) x;
        int i1 = i0 + 1;

        if (y > n + 0.5) y = n + 0.5f;
        if (y < 0.5)     y = 0.5f;

        int j0 = (int) y;
        int j1 = j0 + 1;

        float s1 = x - i0;
        float s0 = 1 - s1;
        float t1 = y - j0;
        float t0 = 1 - t1;

        d[i, j] = s0 * (t0 * d0[i0, j0] + t1 * d0[i0, j1])
                + s1 * (t0 * d0[i1, j0] + t1 * d0[i1, j1]);

      }
    }
    SetBoundry(b, d);
  }


  private void Diffuse(int b, DoubleArray c, DoubleArray c0, float diff)
  {
    float a = dt * diff * n * n;
    LinearSolver(b, c, c0, a, 1 + 4 * a);
  }

  void Project(DoubleArray x, DoubleArray y, DoubleArray p, DoubleArray div)
  {
    for (int i = 0; i < n; ++i)
    { for (int j = 0; j < n; ++j)
      {
        div[i, j] = (x[i+1, j] - x[i-1, j]
                   + y[i, j+1] - y[i, j-1])
                   * - 0.5f / n;
        p[i, j] = 0;
      }
    }

    SetBoundry(0, div);
    SetBoundry(0, p);

    LinearSolver(0, p, div, 1, 4);

    for (int i=0; i<n; ++i)
    { for (int j=0; j<n; ++j)
      {
        x[i, j] -= 0.5f * n * (p[i+1, j] - p[i-1,j]);
        y[i, j] -= 0.5f * n * (p[i, j+1] - p[i, j-1]);
      }
    }

    SetBoundry(1, x);
    SetBoundry(2, y);
  }


  // Iterative linear system solver using the Gauss-sidel
  // relaxation technique.
  void LinearSolver(int b, DoubleArray x, DoubleArray x0, float a, float c)
  {
```

```
    for (int k = 0; k < 15; k++)
    { for (int i=0; i<n; ++i)
      { for (int j=0; j<n; ++j)
        {
          x[i, j] = (a * ( x[i-1, j] + x[i+1, j]
                 +   x[i, j-1] + x[i, j+1])
                 +  x0[i, j]) / c;
        }
      }
      SetBoundry(b, x);
    }
  }


  // Specifies simple boundry conditions.
  // Without this (i.e., if you comment it out), the
  // smoke will just float away - with it, the smoke will
  // be forced back into the region (sort of like a fish-tank)
  private void SetBoundry(int b, DoubleArray x)
  {
    for (int i = 0; i<n; ++i)
    {
      x[  0, i  ] = b == 1 ? -x[1, i] : x[1, i];
      x[n+1, i  ] = b == 1 ? -x[n, i] : x[n, i];
      x[  i, 0  ] = b == 2 ? -x[i, 1] : x[i, 1];
      x[  i, n+1] = b == 2 ? -x[i, n] : x[i, n];
    }

    x[  0,   0] = 0.5f * (x[1, 0  ] + x[  0, 1]);
    x[  0, n+1] = 0.5f * (x[1, n+1] + x[  0, n]);
    x[n+1,   0] = 0.5f * (x[n, 0  ] + x[n+1, 1]);
    x[n+1, n+1] = 0.5f * (x[n, n+1] + x[n+1, n]);
  }

}


class Smoke : System.Windows.Forms.Form
{
  // C# draw update
  private   System.Windows.Forms.Timer  m_timer; // Our update timer

  // Solver variables
  int         m_n  = 220;
  float       m_dt = 0.2f;
  SmokeSolver     m_ss = new SmokeSolver();

  // Mouse Variables
  Point       m_oldmp = new Point(0,0);

  void Reset()
  {
    m_ss.Setup(m_n, m_dt);

    // For testing - draw text onto our start density
    // buffer (use bitmap draw functions)
    #if true
    Bitmap bitmap = new Bitmap( m_n, m_n );
    using(Graphics gb = Graphics.FromImage(bitmap))
    {
      gb.DrawString("Smoke and Gas", new Font("Times", 22, FontStyle.Regular), Brushes.White, 10, 40);
    }

    for (int i = 0; i < bitmap.Width-1; ++i)
    {
      for (int j = 0; j < bitmap.Height-1; ++j)
      {
        Color c = bitmap.GetPixel(i,j);
        if ( c.R > 0 )
        {
          m_ss.d[i,j] = 1.0f;
        }
      }
    }
    #endif
  }

  private void Draw()
  {
    Graphics g    = this.CreateGraphics();
    Rectangle r   = this.ClientRectangle;
    Bitmap bitmap0  = new Bitmap(m_n, m_n);


    // Solve smoke
    m_ss.VelocitySolver();
    m_ss.DensitySolver();

    for (int i=0; i<m_n; ++i)
    { for (int j=0; j<m_n; ++j)
      {
        // Draw density
        if ( m_ss.d[i, j] > 0 )
        {
          int c = (int) ( (1.0 - m_ss.d[i, j]) * 255);
          if (c < 0) c = 0;
          bitmap0.SetPixel(i, j, Color.FromArgb( c, c, c ) );
        }
        else
        {
          bitmap0.SetPixel(i, j, Color.White );
        }
      }
    }

    g.DrawImage(bitmap0, 0, 0, this.Width, this.Height );

    // --- Draw title at top left of screen --- //
    //g.DrawString("Smoke and Gas", this.Font, Brushes.Red, 10, 10);
  }// End of OnPaint(..)

  public void UpdateMouseInput()
  {
    Point mp = System.Windows.Forms.Control.MousePosition;
    mp = this.PointToClient( mp );

    int x = mp.X;
    int y = mp.Y;

    if ( System.Windows.Forms.Control.MouseButtons != System.Windows.Forms.MouseButtons.Left &&
       System.Windows.Forms.Control.MouseButtons != System.Windows.Forms.MouseButtons.Right )
    {
      m_oldmp = mp;
      return;
    }

    // get index for fluid cell under mouse position
    int i = (int) ((x / (float) this.Width)  * m_n + 1);
    int j = (int) ((y / (float) this.Height) * m_n + 1);

    // set boundries
    if (i > m_n)  i = m_n;
    if (i < 1  )  i = 1;
    if (j > m_n)  j = m_n;
    if (j < 1  )  j = 1;

    // Add density
    if ( System.Windows.Forms.Control.MouseButtons == System.Windows.Forms.MouseButtons.Left )
    {
      m_ss.dOld[i, j] = 100;
    }
    // Add velocity
    if ( System.Windows.Forms.Control.MouseButtons == System.Windows.Forms.MouseButtons.Right )
    {
      m_ss.uOld[i, j] = (x - m_oldmp.X) * 5;
      m_ss.vOld[i, j] = (y - m_oldmp.Y) * 5;
    }
  }

  private void OnTimer(object sender, System.EventArgs e)
  {
    Draw();

    UpdateMouseInput();

    if (Control.ModifierKeys == Keys.Control)
```

```
465     {
466       Reset();
467     }
468   }// End of OnTimer(..)
469
470
471   private void InitializeComponent()
472   {
473     // --- Timer --- //
474     this.m_timer = new System.Windows.Forms.Timer↩
        ();
475     this.m_timer.Interval = 50;
476     this.m_timer.Tick += new System.EventHandler(↩
        this.OnTimer);
477     m_timer.Enabled = true;
478
479     // --- Window Title --- //
480     this.Text = "xbdev.net - gases and fluids";
481
482   }// End of InitializeComponent()
483
484   public Smoke()
485   {
486     Reset();
487     InitializeComponent();
488     Width  = 540;
489     Height = 540;
490
491   }// End of Smoke()
492
493   // Our program entry point.
494   static void Main()
495   {
496     System.Windows.Forms.Application.Run(new ↩
        Smoke());
497   }// End of Main()
498 }
```