# I'm doing it wrong

more often than not.

# Why my fluids don't flow

## December 14, 2010

I have an unopened copy of Digital Color Management (http://www.amazon.com/Digital-Color-Management-Wiley--Technology/dp/047051244X/ref=sr_1_1?ie=UTF8&qid=1292301078&sr=8-1) sitting on my desk. It's staring at me accusingly.

In order to keep myself distracted from its dirty looks, I've been tinkering around with fluid simulation. Miles Macklin (http://mmack.wordpress.com/) has done some great work with Eulerian (grid based) solvers, so in an effort to distance myself from the competition, I'm sticking to 2D Lagrangian (particle based) simulation.

Until recently, I'd always thought that particle based fluid simulation was complicated and involved *heavy maths*. This wasn't helped by the fact that most of the papers on the subject have serious sounding names like Particle-based Viscoelastic Fluid Simulation (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.9379&rep=rep1&type=pdf), Weakly compressible SPH for free surface flows (http://cg.informatik.uni-freiburg.de/publications/sphSCA2007.pdf), or even Smoothed Particle Hydrodynamics and Magnetohydrodynamics (http://arxiv.org/abs/1012.1885).

It wasn't until I finally took the plunge and tried writing my own Smoothed Particle Hydrodynamics simulation that I found that it can be quite easy, provided you work from the right papers. SPH has a couple of advantages over grid based methods: it is trivial to ensure that mass is exactly conserved, and free-surfaces (the boundary between fluid and non-fluid) come naturally. Unfortunately, SPH simulations have a tendency to explode if the time step is too large and getting satisfactory results is heavily dependent on finding "good" functions with which to model the inter-particle forces.

I had originally intended to write an introduction to SPH, but soon realised that it would make this post intolerably long, so instead I'll refer to the papers that I used when writing my own sim. Pretty much every SPH paper comes with an introduction to the subject, invariably in section **2. Related Work**.

The first paper I tried implementing was Particle-Based Fluid Simulation for Interactive Applications (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.7720&rep=rep1&type=pdf) by Müller et. al. It serves as a great introduction to SPH with a very good discussion of kernel weighting functions, but I had real difficulty getting decent results. In the paper pressure, viscosity and surface tension forces are modeled using following equations:

$$\mathbf{f}_i^{pressure} = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h)$$

$$\mathbf{f}_i^{viscosity} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h)$$

$$c_S(\mathbf{r}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h)$$

$$\mathbf{f}_i^{surface} = -\sigma \nabla^2 c_S \frac{\mathbf{n}}{|\mathbf{n}|}$$

The pressure for each particle is calculated from its density using:

$P_i = k(\rho_i - \rho_0)$ where $\rho_0$ is the some non-zero rest density.

The first problem I encountered was with the pressure model; it only acts as a repulsive force if the particle density is greater than the rest density. If a particle has only a small number of neighbours, the pressure force will attract them to form a cluster of particles all sharing the same space. In my experiments, I often found large numbers of clusters of three or four particles all in the same position. It took me a while to figure out what was going on because Müller states that the value of the rest density "mathematically has no effect on pressure forces", which is only true given a fairly uniform density of particles far from the boundary.

The second problem I found was with the surface tension force. It was originally developed for multiphase fluid situations with no free surfaces and doesn't behave well near the surface boundary; in fact it can actually pull the fluid into concave shapes. Additionally, because it's based on a Laplacian, it's very sensitive to fluctuations in the particle density, which are the norm at the surface boundary.

After a week or so of trying, this was my best result:

From the outset, you can see the surface tension force is doing weird things. Even worse, once the fluid starts to settle the particles tended to stack on top of each and form a very un-fluid blob.

On the up side, I did create possibly my best ever bug when implementing the surface tension model; I ended up with something resembling microscopic life floating around under the microscope:

The next paper I tried was Particle-based Viscoelastic Fluid Simulation (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.9379&rep=rep1&type=pdf) by Clavet et al. I actually had a lot of success with their paper and had a working implementation of their basic model up and running in less than two hours. Albeit minus the viscoelasticity. In addition to the pressure force described in Müller's paper, they model "near" density and pressure, which are similar to their regular counterparts but with a zero rest density and different kernel functions:

$$P_i = k(\rho_i - \rho_0)$$

$$P_i^{near} = k^{near}\rho_i^{near}$$

$$\rho_i = \sum_j (1 - \frac{|\mathbf{r}_i - \mathbf{r}_j|}{h})^2$$

$$\rho_i^{near} = \sum_j (1 - \frac{|\mathbf{r}_i - \mathbf{r}_j|}{h})^3$$

This near pressure ensures a minimum spacing and as an added bonus performs a decent job of modelling surface tension too. This is the first simulation I ran using their pressure and viscosity forces:

Although initial results were promising, I struggled when tweaking the parameters to find a good balance between a fluid that was too compressible and one that was too viscous. Also, what I really wanted was to do multiphase fluid simulation. This wasn't covered in the viscoelastic paper, so my next port of call was Weakly compressible SPH for free surface flows (cg.informatik.uni-freiburg.de/publications/sphSCA2007.pdf) by Becker et al. In this paper, surface tension is modeled as:

$$\mathbf{f}_i^{surface} = -\frac{\kappa}{m_i} \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j)(\mathbf{r}_i - \mathbf{r}_j)$$

They also discuss using Tait's equation for the pressure force, rather than one based on the ideal gas law:

$$P_i = B((\frac{\rho_i}{\rho_0})^\gamma - 1) \text{ with } \gamma = 7$$

I gave that a shot, but the large exponent caused the simulation to explode unless I used a *really* small time step. Instead, I found that modifying the pressure forces from the viscoelastic paper slightly gave a much less compressible fluid without the requirement for a tiny time step:

$$\rho_i = \sum_j (1 - \frac{|\mathbf{r}_i - \mathbf{r}_j|}{h})^3$$

$$\rho_i^{near} = \sum_j (1 - \frac{|\mathbf{r}_i - \mathbf{r}_j|}{h})^4$$

Here's one of my more successful runs:

And here is a slightly simplified version of the code behind it. Be warned, it's quite messy; I'm rather enjoying hacking code together these days:

```
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <memory.h>
#include <glut.h>


#define kScreenWidth 640
#define kScreenHeight 480
#define kViewWidth 10.0f
#define kViewHeight (kScreenHeight*kViewWidth/kScreenWidth)
#define kPi 3.1415926535f
#define kParticleCount 3000

#define kRestDensity 82.0f
#define kStiffness 0.08f
#define kNearStiffness 0.1f
#define kSurfaceTension 0.0004f
#define kLinearViscocity 0.5f
#define kQuadraticViscocity 1.0f

#define kParticleRadius 0.05f
#define kH (6*kParticleRadius)
#define kFrameRate 20
#define kSubSteps 7

#define kDt ((1.0f/kFrameRate) / kSubSteps)
#define kDt2 (kDt*kDt)
#define kNorm (20/(2*kPi*kH*kH))
#define kNearNorm (30/(2*kPi*kH*kH))

```

```
34
35  #define kEpsilon 0.0000001f
36  #define kEpsilon2 (kEpsilon*kEpsilon)
37
38
39  struct Particle
40  {
41      float x;
42      float y;
43
44      float u;
45      float v;
46
47      float P;
48      float nearP;
49
50      float m;
51
52      float density;
53      float nearDensity;
54      Particle* next;
55  };
56
57  struct Vector2
58  {
59      Vector2() { }
60      Vector2(float x, float y) : x(x) , y(y) { }
61      float x;
62      float y;
63  };
64
65  struct Wall
66  {
67      Wall() { }
68      Wall(float _nx, float _ny, float _c) : nx(_nx), ny(_ny),
69      float nx;
70      float ny;
71      float c;
72  };
73
74  struct Rgba
75  {
76      Rgba() { }
77      Rgba(float r, float g, float b, float a) : r(r), g(g), b
78      float r, g, b, a;
79  };
80
81  struct Material
82  {
83      Material() { }
84      Material(const Rgba& colour, float mass, float scale, fl
85      Rgba colour;
86      float mass;
87      float scale;
88      float bias;
89  };
90
```

```
 91  #define kMaxNeighbourCount 64
 92  struct Neighbours
 93  {
 94      const Particle* particles[kMaxNeighbourCount];
 95      float r[kMaxNeighbourCount];
 96      size_t count;
 97  };
 98
 99  size_t particleCount = 0;
100  Particle particles[kParticleCount];
101  Neighbours neighbours[kParticleCount];
102  Vector2 prevPos[kParticleCount];
103  Vector2 relaxedPos[kParticleCount];
104  Material particleMaterials[kParticleCount];
105  Rgba shadedParticleColours[kParticleCount];
106
107  #define kWallCount 4
108  Wall walls[kWallCount] =
109  {
110      Wall( 1,  0, 0),
111      Wall( 0,  1, 0),
112      Wall(-1,  0, -kViewWidth),
113      Wall( 0, -1, -kViewHeight)
114  };
115
116  #define kCellSize kH
117  const size_t kGridWidth = (size_t)(kViewWidth / kCellSize);
118  const size_t kGridHeight = (size_t)(kViewHeight / kCellSize)
119  const size_t kGridCellCount = kGridWidth * kGridHeight;
120  Particle* grid[kGridCellCount];
121  size_t gridCoords[kParticleCount*2];
122
123
124  struct Emitter
125  {
126      Emitter(const Material& material, const Vector2& positio
127          : material(material), position(position), direction(
128      {
129          float len = sqrt(direction.x*direction.x + direction
130          this->direction.x /= len;
131          this->direction.y /= len;
132      }
133      Material material;
134      Vector2 position;
135      Vector2 direction;
136      float size;
137      float speed;
138      float delay;
139      size_t count;
140  };
141
142  #define kEmitterCount 2
143  Emitter emitters[kEmitterCount] =
144  {
145      Emitter(
146          Material(Rgba(0.6f, 0.7f, 0.9f, 1), 1.0f, 0.08f, 0.9
147          Vector2(0.05f*kViewWidth, 0.8f*kViewHeight), Vector2
```

```
148     Emitter(
149         Material(Rgba(0.1f, 0.05f, 0.3f, 1), 1.4f, 0.075f, 1
150         Vector2(0.05f*kViewWidth, 0.9f*kViewHeight), Vector2
151 };
152 
153 
154 float Random01() { return (float)rand() / (float)(RAND_MAX-1
155 float Random(float a, float b) { return a + (b-a)*Random01()
156 
157 
158 void UpdateGrid()
159 {
160     // Clear grid
161     memset(grid, 0, kGridCellCount*sizeof(Particle*));
162 
163     // Add particles to grid
164     for (size_t i=0; i<particleCount; ++i)
165     {
166         Particle& pi = particles[i];
167         int x = pi.x / kCellSize;
168         int y = pi.y / kCellSize;
169 
170         if (x < 1)
171             x = 1;
172         else if (x > kGridWidth-2)
173             x = kGridWidth-2;
174 
175         if (y < 1)
176             y = 1;
177         else if (y > kGridHeight-2)
178             y = kGridHeight-2;
179 
180         pi.next = grid[x+y*kGridWidth];
181         grid[x+y*kGridWidth] = &pi;
182 
183         gridCoords[i*2] = x;
184         gridCoords[i*2+1] = y;
185     }
186 }
187 
188 
189 void ApplyBodyForces()
190 {
191     for (size_t i=0; i<particleCount; ++i)
192     {
193         Particle& pi = particles[i];
194         pi.v -= 9.8f*kDt;
195     }
196 }
197 
198 
199 void Advance()
200 {
201     for (size_t i=0; i<particleCount; ++i)
202     {
203         Particle& pi = particles[i];
204 
```

```
205             // preserve current position
206             prevPos[i].x = pi.x;
207             prevPos[i].y = pi.y;
208 
209             pi.x += kDt * pi.u;
210             pi.y += kDt * pi.v;
211         }
212     }
213 
214 
215     void CalculatePressure()
216     {
217         for (size_t i=0; i<particleCount; ++i)
218         {
219             Particle& pi = particles[i];
220             size_t gi = gridCoords[i*2];
221             size_t gj = gridCoords[i*2+1]*kGridWidth;
222 
223             neighbours[i].count = 0;
224 
225             float density = 0;
226             float nearDensity = 0;
227             for (size_t ni=gi-1; ni<=gi+1; ++ni)
228             {
229                 for (size_t nj=gj-kGridWidth; nj<=gj+kGridWidth;
230                 {
231                     for (Particle* ppj=grid[ni+nj]; NULL!=ppj; p
232                     {
233                         const Particle& pj = *ppj;
234 
235                         float dx = pj.x - pi.x;
236                         float dy = pj.y - pi.y;
237                         float r2 = dx*dx + dy*dy;
238                         if (r2 < kEpsilon2 || r2 > kH*kH)
239                             continue;
240 
241                         float r = sqrt(r2);
242                         float a = 1 - r/kH;
243                         density += pj.m * a*a*a * kNorm;
244                         nearDensity += pj.m * a*a*a*a * kNearNor
245 
246                         if (neighbours[i].count < kMaxNeighbourC
247                         {
248                             neighbours[i].particles[neighbours[i
249                             neighbours[i].r[neighbours[i].count]
250                             ++neighbours[i].count;
251                         }
252                     }
253                 }
254             }
255 
256             pi.density = density;
257             pi.nearDensity = nearDensity;
258             pi.P = kStiffness * (density - pi.m*kRestDensity);
259             pi.nearP = kNearStiffness * nearDensity;
260         }
261     }
```

```
262
263
264 void CalculateRelaxedPositions()
265 {
266     for (size_t i=0; i<particleCount; ++i)
267     {
268         const Particle& pi = particles[i];
269
270         float x = pi.x;
271         float y = pi.y;
272
273         for (size_t j=0; j<neighbours[i].count; ++j)
274         {
275             const Particle& pj = *neighbours[i].particles[j]
276             float r = neighbours[i].r[j];
277             float dx = pj.x - pi.x;
278             float dy = pj.y - pi.y;
279
280             float a = 1 - r/kH;
281
282             float d = kDt2 * ((pi.nearP+pj.nearP)*a*a*a*kNea
283
284             // relax
285             x -= d * dx / (r*pi.m);
286             y -= d * dy / (r*pi.m);
287
288             // surface tension
289             if (pi.m == pj.m)
290             {
291                 x += (kSurfaceTension/pi.m) * pj.m*a*a*kNorm
292                 y += (kSurfaceTension/pi.m) * pj.m*a*a*kNorm
293             }
294
295             // viscocity
296             float du = pi.u - pj.u;
297             float dv = pi.v - pj.v;
298             float u = du*dx + dv*dy;
299             if (u > 0)
300             {
301                 u /= r;
302
303                 float a = 1 - r/kH;
304                 float I = 0.5f * kDt * a * (kLinearViscocity
305
306                 x -= I * dx * kDt;
307                 y -= I * dy * kDt;
308             }
309
310         }
311
312         relaxedPos[i].x = x;
313         relaxedPos[i].y = y;
314     }
315 }
316
317
318 void MoveToRelaxedPositions()
```

```
319 {
320     for (size_t i=0; i<particleCount; ++i)
321     {
322         Particle& pi = particles[i];
323         pi.x = relaxedPos[i].x;
324         pi.y = relaxedPos[i].y;
325         pi.u = (pi.x - prevPos[i].x) / kDt;
326         pi.v = (pi.y - prevPos[i].y) / kDt;
327     }
328 }
329 
330 
331 void ResolveCollisions()
332 {
333     for (size_t i=0; i<particleCount; ++i)
334     {
335         Particle& pi = particles[i];
336 
337         for (size_t j=0; j<kWallCount; ++j)
338         {
339             const Wall& wall = walls[j];
340             float dis = wall.nx*pi.x + wall.ny*pi.y - wall.c
341             if (dis < kParticleRadius)
342             {
343                 float d = pi.u*wall.nx + pi.v*wall.ny;
344                 if (dis < 0)
345                     dis = 0;
346                 pi.u += (kParticleRadius - dis) * wall.nx / 
347                 pi.v += (kParticleRadius - dis) * wall.ny / 
348             }
349         }
350     }
351 }
352 
353 
354 void Render()
355 {
356     glClearColor(0.02f, 0.01f, 0.01f, 1);
357     glClear(GL_COLOR_BUFFER_BIT);
358 
359     glMatrixMode(GL_PROJECTION);
360     glLoadIdentity();
361     glOrtho(0, kViewWidth, 0, kViewHeight, 0, 1);
362 
363     glEnable(GL_POINT_SMOOTH);
364     glEnable(GL_BLEND);
365     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
366 
367     for (size_t i=0; i<particleCount; ++i)
368     {
369         const Particle& pi = particles[i];
370         const Material& material = particleMaterials[i];
371 
372         Rgba& rgba = shadedParticleColours[i];
373         rgba = material.colour;
374         rgba.r *= material.bias + material.scale*pi.P;
375         rgba.g *= material.bias + material.scale*pi.P;
```

```
376             rgba.b *= material.bias + material.scale*pi.P;
377         }
378 
379         glEnableClientState(GL_VERTEX_ARRAY);
380         glEnableClientState(GL_COLOR_ARRAY);
381 
382         glPointSize(2.5f*kParticleRadius*kScreenWidth/kViewWidth
383 
384         glColorPointer(4, GL_FLOAT, sizeof(Rgba), shadedParticle
385         glVertexPointer(2, GL_FLOAT, sizeof(Particle), particles
386         glDrawArrays(GL_POINTS, 0, particleCount);
387 
388         glDisableClientState(GL_COLOR_ARRAY);
389         glDisableClientState(GL_VERTEX_ARRAY);
390 
391         glutSwapBuffers();
392     }
393 
394 
395     void EmitParticles()
396     {
397         if (particleCount == kParticleCount)
398             return;
399 
400         static int emitDelay = 0;
401         if (++emitDelay < 3)
402             return;
403 
404         for (size_t emitterIdx=0; emitterIdx<kEmitterCount; ++em
405         {
406             Emitter& emitter = emitters[emitterIdx];
407             if (emitter.count >= kParticleCount/kEmitterCount)
408                 continue;
409 
410             emitter.delay -= kDt*emitDelay;
411             if (emitter.delay > 0)
412                 continue;
413 
414             size_t steps = emitter.size / (2*kParticleRadius);
415 
416             for (size_t i=0; i<=steps && particleCount<kParticle
417             {
418                 Particle& pi = particles[particleCount];
419                 Material& material = particleMaterials[particleC
420                 ++particleCount;
421 
422                 ++emitter.count;
423 
424                 float ofs = (float)i / (float)steps - 0.5f;
425 
426                 ofs *= emitter.size;
427                 pi.x = emitter.position.x - ofs*emitter.directio
428                 pi.y = emitter.position.y + ofs*emitter.directio
429                 pi.u = emitter.speed * emitter.direction.x*Rando
430                 pi.v = emitter.speed * emitter.direction.y*Rando
431                 pi.m = emitter.material.mass;
432 
```

```
433                 material = emitter.material;
434             }
435         }
436 
437         emitDelay = 0;
438     }
439 
440 
441     void Update()
442     {
443         for (size_t step=0; step<kSubSteps; ++step)
444         {
445             EmitParticles();
446 
447             ApplyBodyForces();
448             Advance();
449             UpdateGrid();
450             CalculatePressure();
451             CalculateRelaxedPositions();
452             MoveToRelaxedPositions();
453             UpdateGrid();
454             ResolveCollisions();
455         }
456 
457         glutPostRedisplay();
458     }
459 
460 
461     int main (int argc, char** argv)
462     {
463         glutInitWindowSize(kScreenWidth, kScreenHeight);
464         glutInit(&argc, argv);
465         glutInitDisplayString("samples stencil>=3 rgb double dep
466         glutCreateWindow("SPH");
467         glutDisplayFunc(Render);
468         glutIdleFunc(Update);
469 
470         memset(particles, 0, kParticleCount*sizeof(Particle));
471         UpdateGrid();
472 
473         glutMainLoop();
474 
475         return 0;
476     }
```

I’m pretty happy with the results, even if at three seconds per frame for the video above, my implementation isn’t exactly fast. Here are a few other videos from various stages of development:

Posted by Tom Madams
Filed in wrongness
Tags: 2d, fluid simulation, graphics, smoothed particle hydrodynamics, sph
31 Comments »