mikeash.com: just this guy, you know?

Posted at 2006-03-13 00:00 | RSS feed (Full text feed) | Blog Index
Next article: Cocoa SIMBL Plugins
Previous article: NSOpenGLContext and one-shot
Tags: fluid simulation

Home
Book
The Complete Friday Q&A, advanced topics in Mac OS X and iOS programming.
Blog
GitHub
My GitHub page, containing various open-source libraries for Mac and iOS development, and some miscellaneous projects
Glider Flying
HD Ridge Running Video
List of Landouts
Day in the Life
Skyline Soaring Club
Soaring Society of America
Getting Answers
Ten simple points to follow to get good answers on IRC, mailing lists, and other places
Miscellaneous Pages
Miscellaneous old, rarely-updated content
mike@mikeash.com
E-mail me

Fluid Simulation for Dummies

by Mike Ash

In the spring and summer of 2005, I wrote my Master's thesis on high-performance real-time 3D fluid simulation and volumetric rendering. The basics of the fluid simulation that I used are straightforward, but I had a very difficult time understanding it. The available reference materials were all very good, but they were a bit too physics-y and math-y for me. Unable to find something geared towards somebody of my mindset, I'd like to write the page I wish I'd had a year ago. With that goal in mind, I'm going to show you how to do simple 3D fluid simulation, step-by-step, with as much emphasis on the actual programming as possible.

The simulation code and ideas that I'm going to present are based on Jos Stam's paper Real-Time Fluid Dynamics for Games. If you want a more in-depth look at what's going on, that's the place to go. You can also read all about how to parallelize the simulation and render the output in 3D in my Master's thesis.

Basics

Fluid simulation is based on the Navier-Stokes equations, which are fundamental, interesting, and difficult to understand without a good background in physics and differential equations. To that end, I'm going to pretty much ignore them except to very briefly explain what they say.

Think of a fluid as a collection of boxes or cells. Each box has various properties, like velocity and density. These boxes interact with their neighbors, affecting their velocity and density. A real-world fluid can be seen as having a huge number of absolutely miniscule boxes, all interacting with their neighbors a zillion times a second.

Of course, a real computer can't handle a zillion interactions per second, nor can it handle a zillion little boxes, so we have to compromise. We'll break the fluid up into a reasonable number of boxes and try to do several interactions per second.

To make things simpler, we're only going to examine incompressible fluid. Air is an example of a compressible fluid; you squish it and it gets smaller. Water is an example of an incompressible fluid; you squish it and it pushes back, and doesn't get any smaller. Incompressible fluids are simpler to simulate because their density and pressure is always constant.

Of course, moving water is really boring if there's nothing in it, because you can't see it moving! So we'll add some dye so we can actually see it moving around. The water is equally dense everywhere, but some of it has more dye than others, and this variation lets us see things moving. So remember, whenever I'm talking about "density", I'm actually talking about the density of the *dye*, not the density of the *fluid*. (This took me about six months to figure out, that's why I'm so insistent.)

Data Structures

These boxes will be represented as several 3D arrays. Of course, since C hates multidimensional

arrays, we'll use 1D arrays and fake the extra two dimensions. To that end, we have our very first code: a #define which takes three coordinates, and dumps out the corresponding array index in a one-dimensional array:

```
#define IX(x, y, z) ((x) + (y) * N + (z) * N * N)
```

Next, let's actually look at how we represent a cube of fluid. We're going to need its size (this code only handles perfect cubes which have the same length on each size), the length of the timestep, the amount of diffusion (how fast stuff spreads out in the fluid), and the viscosity (how thick the fluid is). We also need a density array and three velocity arrays, one for x, y, and z. We also need scratch space for each array so that we can keep old values around while we compute the new ones.

Putting all of this together, we get our fluid cube structure.

```
struct FluidCube {
    int size;
    float dt;
    float diff;
    float visc;

    float *s;
    float *density;

    float *Vx;
    float *Vy;
    float *Vz;

    float *Vx0;
    float *Vy0;
    float *Vz0;
};
typedef struct FluidCube FluidCube;
```

To make sure everything gets set up properly, we need a function that will fill out this whole structure for us:

```
FluidCube *FluidCubeCreate(int size, int diffusion, int viscosity, float dt)
{
    FluidCube *cube = malloc(sizeof(*cube));
    int N = size;

    cube->size = size;
    cube->dt = dt;
    cube->diff = diffusion;
    cube->visc = viscosity;

    cube->s = calloc(N * N * N, sizeof(float));
    cube->density = calloc(N * N * N, sizeof(float));

    cube->Vx = calloc(N * N * N, sizeof(float));
    cube->Vy = calloc(N * N * N, sizeof(float));
    cube->Vz = calloc(N * N * N, sizeof(float));

    cube->Vx0 = calloc(N * N * N, sizeof(float));
    cube->Vy0 = calloc(N * N * N, sizeof(float));
    cube->Vz0 = calloc(N * N * N, sizeof(float));

    return cube;
}
```

And finally we need to be able to destroy this thing once we're done with it:

```
void FluidCubeFree(FluidCube *cube)
{
    free(cube->s);
    free(cube->density);

    free(cube->Vx);
    free(cube->Vy);
    free(cube->Vz);
```

```
    free(cube->Vx0);
    free(cube->Vy0);
    free(cube->Vz0);

    free(cube);
}
```

Since this just initializes a motionless cube with no dye, we need a way both to add some dye:

```
void FluidCubeAddDensity(FluidCube *cube, int x, int y, int z, float amount)
{
    int N = cube->size;
    cube->density[IX(x, y, z)] += amount;
}
```

And to add some velocity:

```
void FluidCubeAddVelocity(FluidCube *cube, int x, int y, int z, float amountX,
float amountY, float amountZ)
{
    int N = cube->size;
    int index = IX(x, y, z);

    cube->Vx[index] += amountX;
    cube->Vy[index] += amountY;
    cube->Vz[index] += amountZ;
}
```

Simulation Outline

There are three main operations that we'll use to perform a simulation step.

diffuse - Put a drop of soy sauce in some water, and you'll notice that it doesn't stay still, but it spreads out. This happens even if the water and sauce are both perfectly still. This is called diffusion. We use diffusion both in the obvious case of making the dye spread out, and also in the less obvious case of making the velocities of the fluid spread out.

project - Remember when I said that we're only simulating incompressible fluids? This means that the amount of fluid in each box has to stay constant. That means that the amount of fluid going in has to be exactly equal to the amount of fluid going out. The other operations tend to screw things up so that you get some boxes with a net outflow, and some with a net inflow. This operation runs through all the cells and fixes them up so everything is in equilibrium.

advect - Every cell has a set of velocities, and these velocities make things move. This is called advection. As with diffusion, advection applies both to the dye and to the velocities themselves.

There are also two subroutines that are used by these operations.

set_bnd - This is short for "set bounds", and it's a way to keep fluid from leaking out of your box. Not that it could really leak, since it's just a simulation in memory, but not having walls really screws up the simulation code. Walls are added by treating the outer layer of cells as the wall. Basically, every velocity in the layer next to this outer layer is mirrored. So when you have some velocity towards the wall in the next-to-outer layer, the wall gets a velocity that perfectly counters it.

lin_solve - I honestly don't know exactly what this function does or how it works. What I do know is that it's used for both **diffuse** and **project**. It's solving a linear differential equation of some sort, although how and what is not entirely clear to me.

OK, now we're ready to see the main simulation function:

```
void FluidCubeStep(FluidCube *cube)
{
    int N          = cube->size;
```

```
    float visc     = cube->visc;
    float diff     = cube->diff;
    float dt       = cube->dt;
    float *Vx      = cube->Vx;
    float *Vy      = cube->Vy;
    float *Vz      = cube->Vz;
    float *Vx0     = cube->Vx0;
    float *Vy0     = cube->Vy0;
    float *Vz0     = cube->Vz0;
    float *s       = cube->s;
    float *density = cube->density;

    diffuse(1, Vx0, Vx, visc, dt, 4, N);
    diffuse(2, Vy0, Vy, visc, dt, 4, N);
    diffuse(3, Vz0, Vz, visc, dt, 4, N);

    project(Vx0, Vy0, Vz0, Vx, Vy, 4, N);

    advect(1, Vx, Vx0, Vx0, Vy0, Vz0, dt, N);
    advect(2, Vy, Vy0, Vx0, Vy0, Vz0, dt, N);
    advect(3, Vz, Vz0, Vx0, Vy0, Vz0, dt, N);

    project(Vx, Vy, Vz, Vx0, Vy0, 4, N);

    diffuse(0, s, density, diff, dt, 4, N);
    advect(0, density, s, Vx, Vy, Vz, dt, N);
}
```

To summarize, this does the following steps:

- Diffuse all three velocity components.
- Fix up velocities so they keep things incompressible
- Move the velocities around according to the velocities of the fluid (confused yet?).
- Fix up the velocities *again*
- Diffuse the dye.
- Move the dye around according to the velocities.

So that's the basics of it. Now we'll get into the implementation of each function.

set_bnd

As noted above, this function sets the boundary cells at the outer edges of the cube so they perfectly counteract their neighbors.

There's a bit of oddness here which is, what is this `b` pramaeter? It can be 0, 1, 2, or 3, and each value has a special meaning which is not at all obvious. The answer lies is what kind of data can be passed into this function.

We have four different kinds of data (x, y, and z velocities, plus density) floating around, and any of those four can be passed in to `set_bnd`. You may notice that this is exactly the number of values this parameter can have, and this is not a coincidence.

Let's look at a boundary cell (horrible ASCII art warning):

```
+---+---+
|   |^  |
|   |  |
|   |  |
+---+---+
```

Here we're at the left edge of the cube. The left cell is the boundary cell that needs to counteract its neighbor, the right cell. The right cell has a velocity that's up and to the left.

The boundary cell's **x** velocity needs to be *opposite* its neighbor, but its **y** velocity needs to be *equal* to its neighbor. This will produce a result like so:

```
+---+---+
|  ^|^  |
```

```
| / |  |
|/  |  |
+---+---+
```

This counteracts the motion of the fluid which would take it through the wall, and preserves the rest of the motion. So you can see that what action is taken depends on which array we're dealing with; if we're dealing with x velocities, then we have to set the boundary cell's value to the opposite of its neighbor, but for everything else we set it to be the same.

That is the answer to the mysterious b. It tells the function which array it's dealing with, and so whether each boundary cell should be set equal or opposite its neighbor's value.

This function also sets corners. This is done very simply, by setting each corner cell equal to the average of its three neighbors.

Without further ado, the code:

```
static void set_bnd(int b, float *x, int N)
{
    for(int j = 1; j < N - 1; j++) {
        for(int i = 1; i < N - 1; i++) {
            x[IX(i, j, 0  )] = b == 3 ? -x[IX(i, j, 1  )] : x[IX(i, j, 1  )];
            x[IX(i, j, N-1)] = b == 3 ? -x[IX(i, j, N-2)] : x[IX(i, j, N-2)];
        }
    }
    for(int k = 1; k < N - 1; k++) {
        for(int i = 1; i < N - 1; i++) {
            x[IX(i, 0  , k)] = b == 2 ? -x[IX(i, 1  , k)] : x[IX(i, 1  , k)];
            x[IX(i, N-1, k)] = b == 2 ? -x[IX(i, N-2, k)] : x[IX(i, N-2, k)];
        }
    }
    for(int k = 1; k < N - 1; k++) {
        for(int j = 1; j < N - 1; j++) {
            x[IX(0  , j, k)] = b == 1 ? -x[IX(1  , j, k)] : x[IX(1  , j, k)];
            x[IX(N-1, j, k)] = b == 1 ? -x[IX(N-2, j, k)] : x[IX(N-2, j, k)];
        }
    }

    x[IX(0, 0, 0)]       = 0.33f * (x[IX(1, 0, 0)]
                                  + x[IX(0, 1, 0)]
                                  + x[IX(0, 0, 1)]);
    x[IX(0, N-1, 0)]     = 0.33f * (x[IX(1, N-1, 0)]
                                  + x[IX(0, N-2, 0)]
                                  + x[IX(0, N-1, 1)]);
    x[IX(0, 0, N-1)]     = 0.33f * (x[IX(1, 0, N-1)]
                                  + x[IX(0, 1, N-1)]
                                  + x[IX(0, 0, N)]);
    x[IX(0, N-1, N-1)]   = 0.33f * (x[IX(1, N-1, N-1)]
                                  + x[IX(0, N-2, N-1)]
                                  + x[IX(0, N-1, N-2)]);
    x[IX(N-1, 0, 0)]     = 0.33f * (x[IX(N-2, 0, 0)]
                                  + x[IX(N-1, 1, 0)]
                                  + x[IX(N-1, 0, 1)]);
    x[IX(N-1, N-1, 0)]   = 0.33f * (x[IX(N-2, N-1, 0)]
                                  + x[IX(N-1, N-2, 0)]
                                  + x[IX(N-1, N-1, 1)]);
    x[IX(N-1, 0, N-1)]   = 0.33f * (x[IX(N-2, 0, N-1)]
                                  + x[IX(N-1, 1, N-1)]
                                  + x[IX(N-1, 0, N-2)]);
    x[IX(N-1, N-1, N-1)] = 0.33f * (x[IX(N-2, N-1, N-1)]
                                  + x[IX(N-1, N-2, N-1)]
                                  + x[IX(N-1, N-1, N-2)]);
}
```

lin_solve

As stated before, this function is mysterious, but it does some kind of solving. this is done by running through the whole array and setting each cell to a combination of its neighbors. It does this several times; the more iterations it does, the more accurate the results, but the slower

things run. In the step function above, four iterations are used. After each iteration, it resets the boundaries so the calculations don't explode.

Here's the code:

```
static void lin_solve(int b, float *x, float *x0, float a, float c, int iter, int
N)
{
    float cRecip = 1.0 / c;
    for (int k = 0; k < iter; k++) {
        for (int m = 1; m < N - 1; m++) {
            for (int j = 1; j < N - 1; j++) {
                for (int i = 1; i < N - 1; i++) {
                    x[IX(i, j, m)] =
                        (x0[IX(i, j, m)]
                            + a*(    x[IX(i+1, j  , m  )]
                                    +x[IX(i-1, j  , m  )]
                                    +x[IX(i  , j+1, m  )]
                                    +x[IX(i  , j-1, m  )]
                                    +x[IX(i  , j  , m+1)]
                                    +x[IX(i  , j  , m-1)]
                            )) * cRecip;
                }
            }
        }
        set_bnd(b, x, N);
    }
}
```

diffuse

Diffuse is really simple; it just precalculates a value and passes everything off to `lin_solve`. So that means, while I know what it does, I don't really know how, since all the work is in that mysterious function. Code:

```
static void diffuse (int b, float *x, float *x0, float diff, float dt, int iter,
int N)
{
    float a = dt * diff * (N - 2) * (N - 2);
    lin_solve(b, x, x0, a, 1 + 6 * a, iter, N);
}
```

project

This function is also somewhat mysterious as to exactly how it works, but it does some more running through the data and setting values, with some calls to `lin_solve` thrown in for fun. Code:

```
static void project(float *velocX, float *velocY, float *velocZ, float *p, float
*div, int iter, int N)
{
    for (int k = 1; k < N - 1; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {
                div[IX(i, j, k)] = -0.5f*(
                         velocX[IX(i+1, j  , k  )]
                        -velocX[IX(i-1, j  , k  )]
                        +velocY[IX(i  , j+1, k  )]
                        -velocY[IX(i  , j-1, k  )]
                        +velocZ[IX(i  , j  , k+1)]
                        -velocZ[IX(i  , j  , k-1)]
                    )/N;
                p[IX(i, j, k)] = 0;
            }
        }
    }
    set_bnd(0, div, N);
    set_bnd(0, p, N);
    lin_solve(0, p, div, 1, 6, iter, N);
```

```
    for (int k = 1; k < N - 1; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {
                velocX[IX(i, j, k)] -= 0.5f * (  p[IX(i+1, j, k)]
                                                -p[IX(i-1, j, k)]) * N;
                velocY[IX(i, j, k)] -= 0.5f * (  p[IX(i, j+1, k)]
                                                -p[IX(i, j-1, k)]) * N;
                velocZ[IX(i, j, k)] -= 0.5f * (  p[IX(i, j, k+1)]
                                                -p[IX(i, j, k-1)]) * N;
            }
        }
    }
    set_bnd(1, velocX, N);
    set_bnd(2, velocY, N);
    set_bnd(3, velocZ, N);
}
```

advect

This function is responsible for actually moving things around. To that end, it looks at each cell in turn. In that cell, it grabs the velocity, follows that velocity back in time, and sees where it lands. It then takes a weighted average of the cells around the spot where it lands, then applies that value to the current cell.

Here's the code:

```
static void advect(int b, float *d, float *d0,  float *velocX, float *velocY, float
*velocZ, float dt, int N)
{
    float i0, i1, j0, j1, k0, k1;

    float dtx = dt * (N - 2);
    float dty = dt * (N - 2);
    float dtz = dt * (N - 2);

    float s0, s1, t0, t1, u0, u1;
    float tmp1, tmp2, tmp3, x, y, z;

    float Nfloat = N;
    float ifloat, jfloat, kfloat;
    int i, j, k;

    for(k = 1, kfloat = 1; k < N - 1; k++, kfloat++) {
        for(j = 1, jfloat = 1; j < N - 1; j++, jfloat++) {
            for(i = 1, ifloat = 1; i < N - 1; i++, ifloat++) {
                tmp1 = dtx * velocX[IX(i, j, k)];
                tmp2 = dty * velocY[IX(i, j, k)];
                tmp3 = dtz * velocZ[IX(i, j, k)];
                x    = ifloat - tmp1;
                y    = jfloat - tmp2;
                z    = kfloat - tmp3;

                if(x < 0.5f) x = 0.5f;
                if(x > Nfloat + 0.5f) x = Nfloat + 0.5f;
                i0 = floorf(x);
                i1 = i0 + 1.0f;
                if(y < 0.5f) y = 0.5f;
                if(y > Nfloat + 0.5f) y = Nfloat + 0.5f;
                j0 = floorf(y);
                j1 = j0 + 1.0f;
                if(z < 0.5f) z = 0.5f;
                if(z > Nfloat + 0.5f) z = Nfloat + 0.5f;
                k0 = floorf(z);
                k1 = k0 + 1.0f;

                s1 = x - i0;
                s0 = 1.0f - s1;
                t1 = y - j0;
                t0 = 1.0f - t1;
```

```
                u1 = z - k0;
                u0 = 1.0f - u1;

                int i0i = i0;
                int i1i = i1;
                int j0i = j0;
                int j1i = j1;
                int k0i = k0;
                int k1i = k1;

                d[IX(i, j, k)] =

                    s0 * ( t0 * (u0 * d0[IX(i0i, j0i, k0i)]
                                 +u1 * d0[IX(i0i, j0i, k1i)])
                        +( t1 * (u0 * d0[IX(i0i, j1i, k0i)]
                                 +u1 * d0[IX(i0i, j1i, k1i)])))
                   +s1 * ( t0 * (u0 * d0[IX(i1i, j0i, k0i)]
                                 +u1 * d0[IX(i1i, j0i, k1i)])
                        +( t1 * (u0 * d0[IX(i1i, j1i, k0i)]
                                 +u1 * d0[IX(i1i, j1i, k1i)])));
            }
        }
    }
    set_bnd(b, d, N);
}
```

Wrapping Up

And there you have it, everything you need to make your own 3D fluid simulation.

View the full source

Note that actually *displaying* this animation is a bit more difficult. This was actually the primary topic of my thesis, as the fluid simulation was pretty much just borrowed from Stam and pulled into 3D. This kind of three-dimensional grid data is not something that most graphics hardware and APIs know how to render, and so it's a fairly challenging topic.

If you want to show the results of the simulation, you have several choices. One would be to rewrite the simulator in two dimensions instead of three, and you can then just display the density pointer onscreen as a regular image. Another choice would be to run the full 3D simulation but only display 2D slice of it. And finally, another choice would be to run a simulation small enough that you can just draw one polygon per cell without taking too much of a speed hit.

Of course, if you like a challenge, feel free to check out the various techniques for volumetric rendering.

I hope you enjoyed this whirlwind tour through fluid simulation. Have fun!

Comments:

N382AF at 2006-03-15 01:01:00:
Nice work here, I saved it for reading later. A bit over my head, but I love learning this stuff!

Thanks!

Artur W at 2006-07-12 14:23:00:
Thank you very much! Finally something for a dummy like myself. Less equations and more detail on turning them into actual implementation.
This helps!

neil at 2006-08-16 08:54:00:
im a newbei and i want to learn more about simulation and animation, but my problem is i cant

afford to enroll to schools, thats why im looking for free ebooks and tutorials which is available in this world. but my collections are not enough for my thirsty head. all i want is to learn more, if anyone could share their knowledge and books or sites i dont know yet, please enlighten me, by the way im making some simulation in physics principles or topics to help me in my carrier as future physics teacher, but i still need to learn lots of things, so guys please share your informations i assure you it wont go to waste. thanks. heres my add, kaerfmove@yahoo.com, thanks!

Brian Silva at 2006-12-02 18:48:00:

Nice work! I am doing exactly the same work with Stam's implementation, but I was having trouble getting the simulation step to function correctly. I took your methodology from the FluidCubeStep and it solved my problem. So, thank you!

I agree with you about the original sources: they are difficult to understand and don't explain the math very well. I'm not entirely mathematically-inclined so I struggled for a long time to understand it. I'm setting out on a school project to explain the math from this particular implementation in a clear way. If you're interested, I can send it to you when I finish.

Again, great work!

mikeash at 2006-12-12 23:27:00:

Brian, I'm glad you found it useful, and please do send me your results if you'd like.

anonymous friend at 2008-01-25 03:07:26:

The lin_solve function simply solve a system of linear equations. Since the matrix of this system is sparse (i.e contain a lot of zeros), this system can be solved efficiently (note that there exist more accurate algorithms) using a Gauss-Seidel algorithm.

What do we solve exactly ? For the diffusion, we try to find tthe densities which, when diffused backward in time, gives the density we started with. In the case of the projection, it is a little more complicated. We are solving a linear system called a poisson equation. In mathematics, Poisson's equation is a partial differential equation. To make sure that the fluid is incompressible, we need to satisfy the non-divergence condition.

The divergence of a vector field (the velicity of the fluid)at a particular location is a scalar value that essentially represents how much the field is 'expanding' at the point. A positive value represents a divergence and a negative value represents a convergence or compression.If we use the assumption that a fluid is incompressible, then that implies that the divergence of the velocity must be zero everywhere.

A useful splitting method for solving the incompressible flow equations is the so-called 'projection method'. In this technique, one first computes a divergent vector field that results from the advection & viscosity terms, and then projects that entire vector field onto the space of non-divergent vector fields
This can be thought of as creating an instantaneous pressure field whose gradient compensates for any divergence caused by the advection & viscosity.

Getting back to the poisson equation, we create these beast to solve the unknown pressure field. The gradient of this pressure field is an additional force on the fluid and the divergence of this gradient compensates for the divergence resulting from the convection-viscosity equation. The result is a final velocity field that has no divergence.

Setis at 2008-06-16 20:40:59:

About the rendering:

Since you know how to take a 2d slice, showing the density of ink, you can render it in 3d this way:

Take n 2d slices in a given time. Draw the slices in textures, so the more ink in a given cell, the less transparent and more colored is the corresponding pixel. Now draw a 3D cube of all the slices stacking. That way, you can see the ink moving in the cube. Just add some code for rotating the cube and so.

Ivan DeWolf at 2008-09-17 10:25:18:

here is another website with a really legible java implementation in 2d... be sure to look at the source code...
it also uses a gauss-seidell relaxation scheme for it's linear solver. I really wish I understood the linear solver part more.

http://www.multires.caltech.edu/teaching/demos/java/stablefluids.htm

Lars at 2008-12-17 05:12:52:
Thanks very much for the 3D conversion of Jos Stam's 2D code -- it has saved me work and brainache. :-)
I was hoping to make a fun screensaver and this should give me a good boost.

nabeel at 2009-04-03 16:20:54:
Great stuff. Wonder why they never make sound this easy in texts. wonder if i can work out mixing in two phases with this???

Tom at 2009-04-14 02:15:33:
Why is there no main function?
actually i know little about c, but i wanna learn this application and apply it into processing.

mikeash at 2009-04-14 02:33:14:
There is no main function because it's a discussion of techniques, not a full working example. You will have to incorporate what is provided into your own code. If you know little C then an application of this complexity is probably not a good place to start. You need to learn to walk before you can run.

student at 2009-05-10 06:36:10:
Im around this same code the 3d code you provide was very helpfull thanks.
Im hava used several linear solvers gauss seidel relaxation (seq and parallel - Red Black), jacobi (seq and parallel) ....
I'm trying to implement the CG method also because I adjusted the stams code to allow internal boundaries (moving or not), among other things.
I have a pertinent question: I understood after considereble digging that we are solving a system of linear equations in the form Ax=b however what confuses me is what is A exactly?

Any comments or regards on this would be quitte helpfull anyone?

responses to: a14722@ubi.pt

willem at 2009-05-14 06:34:43:
great to see it written for a non-mathematician. thanks

Kent at 2009-07-23 15:38:22:
Thanks heaps for posting this code. I was just about to start moving Stams code from 2D to 3D myself. And I really like the way you have presented the code here. Nice work!

fluid at 2009-10-09 10:33:03:
really cool fluid simulation is here: http://www.escapemotions.com/experiments/fluid_fire_3/index.html

uma sankar pradhan at 2010-04-02 22:13:04:
the code is great.
But in the projection method,
during the calculation of divergence of velocity field,why your dividing it by N.I suppose ,it shoud be multiplied as dx=dy=dz=1/N

```
div[IX(i, j, k)] = -0.5f*(
                     velocX[IX(i+1, j , k )]
                    -velocX[IX(i-1, j , k )]
                    +velocY[IX(i , j+1, k )]
                    -velocY[IX(i , j-1, k )]
                    +velocZ[IX(i , j , k+1)]
                    -velocZ[IX(i , j , k-1)]
                  )*N;
```

instead of

```
div[IX(i, j, k)] = -0.5f*(
                     velocX[IX(i+1, j , k )]
                    -velocX[IX(i-1, j , k )]
                    +velocY[IX(i , j+1, k )]
                    -velocY[IX(i , j-1, k )]
                    +velocZ[IX(i , j , k+1)]
                    -velocZ[IX(i , j , k-1)]
                  )/N;
```

One more thing,
the Vx0,Vy0 need to be initialised to zero before each iteration.
I may be wrong.If i am wrong,kindly inform me.

Aaron Moffatt at 2010-06-23 09:01:50:
Very, very useful document. Just to the point, outlining and giving example code of the key concepts. Really appreciate you writing and posting this.

Akash Verma at 2010-10-23 21:31:08:
Could someone please put up a main() for this to see it working?

libero at 2011-02-04 21:45:29:
does anyone have a working main() for this code because seems to be crashing and I cant find the reason for that? any help you be appreciated.

Martin at 2011-03-29 15:48:51:
I had a deep look at the advect function and ask myself if it's correct to call it with the N-value. Shouldn't this be N-2 (or Nflaot be initialized by N-2)?

Anyway, thanks a lot for the code, it's really valuable for learning how this stuff works!

bob at 2011-07-19 18:24:17:
Why do you allocate N*N*N size for each dimension? This becomes impractical when work with large amounts of particles. For example,
sizeof(float) * 8 callocs * (1000 particles ^ 3) = 32 gb.

EWD at 2014-05-31 18:42:54:
Can anybody explain to me why in 2006 people still use variable names such as "b", "N" or "p"??? Are they trying to save the compiler the work with reading long words?? Were the variables called 'shit' and 'fuck', it would still be more helpful!

But reading some of the comments here, some people deserve it.

bob: Yeah, N*N*N elements for a 3-dimensional array is a terrible waste. Try to allocate N elements for a 3-dimensional array.

Scott at 2015-08-23 23:47:11:
Mike, Do you remember why you need to multiply 'a' by '(N-2)^2 ' in the diffuse function? For example Stam does this in 2D, so I'm expecting it to be '(N-2)^3' for 3D?

Sam at 2017-12-07 04:33:33:
I haven't read the whole thing yet, but the pointers are confusing me a bit. Why initialize a pointer rather than a value?

Eli Maynard at 2019-03-24 13:48:08:
Hi, in

```
x[IX(0, 0, N-1)] = 0.33f * (x[IX(1, 0, N-1)]
                            + x[IX(0, 1, N-1)]
                            + x[IX(0, 0, N)]);
```

should the `IX(0, 0, N)` be `IX(0, 0, N-2)`?

Thanks for the article!

Dimitri at 2020-10-26 20:05:42:
Hey, great work!
Truly life-saving when this is not your field of work or studies, feels good to see the implementation beyond the theory
Thanks a lot for posting it

rich engle at 2021-01-14 03:07:01:
is there any way to have 2 fluids, like water and air?

Polaris at 2021-03-03 20:27:03:
Hey are you sure this is for dummies? As a dummy myself, I regret to inform that I was unable to understand.

Aadhavan P at 2021-09-04 16:26:13:
Hi!
Very good article. I'm saving this for later. Will try to implement it after my exams. I'm very interested in simulating stuff like these.

Anssi at 2022-01-10 20:22:55:
I think there are many intances in the code (also in Stam's original article), where 'N', or 'N-2' should be replace by '1/SCALE'.

This because SCALE represents the spatial step size, not 1/(N-2). Right?

For example, the variable 'a' in the diffuse function should be defined as
a=dt*diff*SCALE^{-2}, instead of a=dt*diff*(N-2)^2.

Similar corrections seem to be in order in 'advect' and 'project' functions.

At least this works for me. Now the qualitative local behaviour does not depend on the size of the grid N, as it should not. You need to adjust the parameters again, if you had adjusted them

according to the original code.

Btw. In diffuse function the 'a' is usually less than 1, and we can replace the 'lin_solve' there with the faster (and more straightforward)
x[IX(i, j)] = x0[IX(i, j)] + a*(x0[IX(i+1, j)]+x0[IX(i-1, j)] +x0[IX(i, j+1)]+x0[IX(i, j-1)]-4*x0[IX(i, j)])
which would be stable in this case. (This was considered in Stam's article but discarded unnecessarily in my mind.)

bentheone at 2022-02-15 09:04:28:

@Anssi, yes ! Thank you, I can't believe nobody pointed that out before. The diffusion ratio being dependent on the grid size makes ZERO sense.

Comments RSS feed for this page

Add your thoughts, post a comment:

Spam and off-topic posts will be deleted without notice. Culprits may be publicly humiliated at my sole discretion.

Name:

The Answer to the Ultimate Question of Life, the Universe, and Everything?

Comment:

Formatting: <i> <b> <blockquote> <code>.

NOTE: Due to an increase in spam, URLs are forbidden! Please provide search terms or fragment your URLs so they don't look like URLs.

Preview Post Comment

Code syntax highlighting thanks to Pygments.
Hosted at DigitalOcean.