Simulation Assignment

Eimear Crotty
Bournemouth University

ABSTRACT

A report detailing the implementation of a 2D fluid simulation using the Marker and Cell (MAC) Grid method. The code which is described in this report can be found at https://github.com/eimearc/simulation.

1 BACKGROUND

Apart from the class notes, I referred to Cline et al. (2005), Bridson and Müller-Fischer (2007) and Braley and Sandu (2009).

2 NAVIER-STOKES EQUATIONS

These are the set of equations that define the velocity field of fluid. They specify both the incompressibility of a fluid (equation 1) and how the velocity field changes over time (equation 2).

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\nabla \cdot \boldsymbol{u})\boldsymbol{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \boldsymbol{u} + \boldsymbol{F} \tag{2}$$

3 MAC GRID

The MAC grid method, first proposed in Harlow and Welch (1965), discretizes the simulation space into a grid structure. The grid stores different quantities at different locations. Pressure is stored in the center of each grid cell, while the velocities are stored on the edges of each cell. The x velocity is sampled and stored at the minimum x edge, while the y velocity is sampled and stored at the minimum y edge.

For implementation purposes the grid is composed of three separate 2D arrays (a vector of vectors in C++). One for pressure, *m_pressure*, one for the u_x velocity, *m_x*, and one for the u_y velocity, *m_y*. This leads to greater stability during the pressure solve stage. The number of columns in *m_x* is $resolution + 1$, while the number of rows in *m_y* is $resolution + 1$.

4 ALGORITHM

Time Step

First, we solve for the time step. As we do not want a particle to travel a distance further than the width of a grid cell (h), we use the following CFL equation to find a suitable time step:

$$\Delta t = \frac{h}{|\boldsymbol{u}_{max}|} \tag{3}$$

Grid Update

The grid is then updated to have each cell reflect its type. This is implemented as a 2D array of TYPEs, where TYPE is an *enum* of {FLUID|SOLID|AIR}. Border cells are set to SOLID. If a cell has a fluid particle in it, its TYPE is FLUID, otherwise it is AIR.

Convection

We use a backwards particle trace to move velocities through the field. We only convect components that border a fluid cell. For each velocity component, we use the traceParticle method (see below) to find the velocity of where this particle was previously and update this velocity to the new velocity. This is known as a semi-Lagrangian method, as we are generating a particle and moving it through the velocity field. However, we never draw the particle as it is only used for convection purposes. The Eigen library (Guennebaud et al. (2010)) is used for the sparse matrix calculations.

```
Velocity MAC::traceParticle(const Position &p, float _time)
{
    // Trace particle from point (_x, _y) using RK2.
    Velocity half_prev_v = velocityAtPosition(
                p-(0.5f*_time*velocityAtPosition(p))
    );
    return velocityAtPosition(
        p-_time*(
            velocityAtPosition(p - _time*half_prev_v)
        )
    );
}
```

Gravity

We then apply the gravity vector to each velocity component that borders a fluid cell.

$$\boldsymbol{u}_{new} = \boldsymbol{u} - \Delta t \cdot (0, 9.80665) \tag{4}$$

Viscosity

We apply viscosity to each velocity component that borders a fluid cell.

$$\boldsymbol{u}_{new} = \boldsymbol{u} + \Delta t \nu \nabla^2 \boldsymbol{u} \tag{5}$$

```
void MAC::applyViscosity(float _time)
{
    MAC tmp(m_resolution);
    tmp.m_x = m_x;
    tmp.m_y = m_y;

    Index index;
    for (index.row=0;index.row<=int(m_resolution);++index.row)
    {
        for (index.col=0;index.col<=int(m_resolution);++index.col)
```

```
        {
            if (bordersFluidCellX(index))
            {
                float l = laplacian(index, _time, Dimension::x);
                tmp.m_x[index.row][index.col] += l;
            }
            if (bordersFluidCellY(index))
            {
                float l = laplacian(index, _time, Dimension::y);
                tmp.m_y[index.row][index.col] += l;
            }
        }
    }

    m_x=tmp.m_x;
    m_y=tmp.m_y;
}


float MAC::laplacian(Index index, float time, Dimension dimension)
{
    float l = 0.0f;
    std::vector<std::vector<float>> *pm;
    std::function<bool(Index)> bordersFluidCell;

    switch (dimension)
    {
    case Dimension::x :
        pm = &m_x;
        bordersFluidCell = [&](Index i)
        {
            return this->bordersFluidCellX(i);
        };
        break;
    case Dimension::y :
        pm = &m_y;
        bordersFluidCell = [&](Index i)
        {
            return this->bordersFluidCellY(i);
        };
    }
    const std::vector<std::vector<float>> &m = *pm;

    float x1=0.0f,x2=0.0f;
    float y1=0.0f,y2=0.0f;

    int row=index.row;
    int col=index.col;
    if (bordersFluidCell({row, col-1})) x1=m[row][col-1];
    if (bordersFluidCell({row, col+1})) x2=m[row][col+1];
    if (bordersFluidCell({row-1, col})) y1=m[row-1][col];
    if (bordersFluidCell({row+1, col})) y2=m[row+1][col];

    l = x1 + x2 + y1 + y2;

    l = time*VISCOSITY*l;

    return l;
}
```

Pressure Solve

The velocity field in its current state does not satisfy the incompressibility requirement of the Navier-Stokes equations. We generate a large sparse matrix $\boldsymbol{A}$ and a vector of divergences $\boldsymbol{b}$ and solve the linear equation to get the vector of pressures $\boldsymbol{p}$ that will make our velocity field divergence-free.

$$\boldsymbol{A}\boldsymbol{p} = \boldsymbol{b} \tag{6}$$

Matrix $\boldsymbol{A}$ is generated as follows:

- Each fluid cell i has a row.
- For each fluid cell i, find the number of non-solid neighbours of cell i. Set $\boldsymbol{A}_{ii}$ to be minus this number.
- For each fluid cell i, for each fluid neighbour j, set $\boldsymbol{A}_{ij}$ to 1.

Vector $\boldsymbol{b}$ is generated using the following equation:

$$b_i = \frac{\rho h}{\Delta t}(\nabla \cdot \boldsymbol{u}_i) - k_i p_{atm} \tag{7}$$

where ρ is the density and h is the cell width. $\nabla \cdot \boldsymbol{u}_i$ is a modified form of the divergence (see below), where velocities between fluid and solid cells are zero. k_i is the number of air cells neighbouring cell i and p_{atm} is the atmospheric pressure.

```
auto A = constructCoefficientMatrix();
auto b = constructDivergenceVector(_time);
Eigen::SimplicialCholesky<Eigen::SparseMatrix<double>> solver;
solver.compute(A);
Eigen::VectorXd p = solver.solve(b);

float MAC::calculateModifiedDivergence(size_t _row, size_t _col)
{
    const int &row = _row;
    const int &col = _col;
    // Velocity components between fluid cells and solid cells
    // are considered to be zero.
    float x1=0.0f,y1=0.0f,x2=0.0f,y2=0.0f;
    Index index = {row,col};
    if (!bordersSolidCellY(index)) y1 = m_y[row][col];
    if (!bordersSolidCellX(index)) x1 = m_x[row][col];

    index={row,col+1};
    if (!bordersSolidCellX(index)) x2 = m_x[row][col+1];
    index = {row+1,col};
    if (!bordersSolidCellY(index)) y2 = m_y[row+1][col];

    float xDiv = x2-x1;
    float yDiv = y2-y1;

    return xDiv + yDiv;
}
```

Apply Pressure

Pressure is only applied to velocity components bordering fluid cells but not solid cells. For each such velocity component, we apply pressure using the following equation:

$$\boldsymbol{u}_{new}(x,y) = \boldsymbol{u}(x,y) - \frac{\Delta t}{\rho h}\nabla p(x,y) \tag{8}$$

Note that the pressure has to be interpolated for each x and y velocity component, as these are sampled at different points to the pressure. See below for the implementation of *applyPressure*.

```
void MAC::applyPressure(float _time)
{
    MAC tmp(m_resolution);
    tmp.m_x = m_x;
    tmp.m_y = m_y;

    Index index;
    for (index.row = 0; index.row <= int(m_resolution); ++index.row)
    {
        for (index.col = 0; index.col <= int(m_resolution); ++index.col)
        {
            if (bordersFluidCellX(index)
                && !(bordersSolidCellX(index)))
```

```
            {
                Velocity v = applyPressureToPoint(
                    index, _time, Dimension::x);
                tmp.m_x[index.row][index.col] = v.m_x;
            }

            if (bordersFluidCellY(index)
                && !(bordersSolidCellY(index)))
            {
                Velocity v = applyPressureToPoint(
                    index, _time, Dimension::y);
                tmp.m_y[index.row][index.col] = v.m_y;
            }
        }
    }

    m_x = tmp.m_x;
    m_y = tmp.m_y;
}

Velocity MAC::applyPressureToPoint(const Index &index,
    float _time, Dimension dimension)
{
    Velocity v;
    Velocity gradient;
    float density=0.0f;
    const int &row = index.row;
    const int &col = index.col;

    if (dimension==Dimension::x)
    {
        v = {m_x[row][col],0};
        float x1 = (m_pressure[row][col]
            + m_pressure[row][col-1])/2.0f;
        float x2 = (m_pressure[row][col-1]
            + m_pressure[row][col-2])/2.0f;
        gradient = {x1-x2,0.0f}; // Backwards difference.
        density=(m_density[row][col-1]
            + m_density[row][col])/2.0f;
    }
    else if (dimension==Dimension::y)
    {
        v = {0,m_y[row][col]};
        float y1 = (m_pressure[row][col]
            + m_pressure[row-1][col])/2.0f;
        float y2 = (m_pressure[row-1][col]
            + m_pressure[row-2][col])/2.0f;
        gradient = {0.0f,y1-y2}; // Backwards difference.
        density=(m_density[row-1][col]
            + m_density[row][col])/2.0f;
    }

    auto rhs = (_time/(density*cellWidth))*gradient;
    Velocity result = v-rhs;
    return result;
}
```

Set Border Velocities

In order to ensure that particles don't move into solid cells, we ensure the velocity components bordering solid cells point out of the solid cells, or are zero.

Move Marker Particles

Finally, we move the tracker particles through the vector field using the new velocities. If a cell contains less than one particle, this cell is considered to be an air cell and the particle is influenced by gravity.

5 ANALYSIS

The time step and resolution of MAC grid are important in influencing the overall quality of the simulation. With a low resolution of 5 x 5, for example, we begin to see particles jumping around the simulation incorrectly. This is due to to the low resolution and the noise of the velocity interpolation.

In addition to this, the pressure solve stage becomes unstable when all of the cells within the border fill with fluid. Increasing the cell count increases the stability of the simulation. I have found that a 30 x 30 grid results in a good trade off between the time taken for a simulation step and the resulting appeal of the simulation. In a perfect simulation, new air cells would be generated dynamically when needed to prevent such a situation arising. This is described in Cline et al. (2005).

When increasing the number of particles, the simulation slows down. If this happens, the simulation can be run as a batch job, instead of at render time. The simulation can run in real time on my machine (MacBook Pro, 2013, 2.7 GHz Dual-Core Intel Core i5, 16 GB 1867 MHz DDR3) up to about 6000 particles at a grid resolution of 50 x 50.

6 FUTURE IMPROVEMENTS

I have authored my code to make the switch from 2D to 3D relatively simple, using *typedef*s for *Velocity* and *Position* types. Another improvement would be to move from a static MAC grid to a dynamic one, where we don't need to iterate over all cells, when only a small number of them are occupied by fluid. This approach only stores cells where there are fluid particles, and includes a buffer of air cells around these fluid cells.

My approach for identifying and advecting "single" fluid particles is not sufficiently robust. A better solution would be to track the surface of the water and treat anything above that surface as a drop of water, influenced only by gravity until it rejoins the pool of water.

Multithreading can be implemented during the intermediate steps when building vectors for the pressure solve, or when building the arrays for drawing.

Finally, to make the fluid truly incompressible, any overlapping particles in the simulation should be redistributed. Currently, the count of particles per cell (and, as a result, the density of the cell) doesn't do anything to ensure a relatively even distribution of particles through the cell.

REFERENCES

Braley, C. and Sandu, A., 2009. Fluid simulation for computer graphics: A tutorial in grid based and particle based methods.

Bridson, R. and Müller-Fischer, M., 2007. Fluid simulation: Siggraph 2007 course notesvideo files associated with this course are available from the citation page. *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, New York, NY, USA. Association for Computing Machinery, 1–81.

Cline, D., Cardon, D. A., Egbert, P. K. and Young, B., 2005. Fluid flow for the rest of us : Tutorial of the marker and cell method in computer graphics.

Guennebaud, G., Jacob, B. and others, , 2010. *Eigen v3* [online].

Harlow, F. H. and Welch, J. E., 1965. Numerical calculation of time dependent viscous incompressible flow of fluid with free surface.