流体 2D 模拟渲染

杨俊 张雯 王嘉晨

摘要：快速实时渲染对于计算机图形学再现真实世界是十分关键的问题。本文将研究流体模拟为切入口，基于物理流体力学通过分布式对粒子系统进行快速实时的流体 2D 模拟渲染。并且本文将展示球型火焰和涡流式爆炸两种效果。

关键词：流体力学；分布式；粒子系统；

一、背景

计算机图形学领域一直试图利用计算机来再现真实世界的样子。基于物理的计算机动画也成了一大研究热点。本文主要探讨的基于物理的流体 2D 模拟，涉及的现象包括烟雾、波浪、气泡、火焰以及爆炸等等。这些现象的模拟在计算流体力学中也是热点话题。流体力学强调对问题本身的精确性的求解，给出具体合理的分析，而计算机图形学致力于真实再现流动的视觉效果。对于某些现象，流体力学的研究者们还没给出很好的计算模型来描述视觉效果。而且计算机图形学，计算的快速性实时性比计算精度更加重要。小的计算代价可以更加快速地调整生成画面。由此我们使用分布式来加快流体的模拟，形成更真实更快捷的视觉效果。

二、算法

2.1 几个定义

流体域以规则的网格分解，每个粒子包含相应空间位置的密度和速度，从而定义向量速度场 U 和密度标量场 D。

添加速度力：计算外力对速度场的贡献。

扩散速度：计算速度场的扩散效应。

平流速度：计算受速度场本身影响的速度运动。

发射速度：计算质量守恒对速度场的影响。

添加密度源：计算外部密度源对密度场的贡献。

扩散密度：计算扩散对密度场的影响。

平流密度：计算受速度场影响的密度场的运动。

2.2 具体算法

在计算机上映射稳定流体需要将流体场分解成多个平铺的子域，并将核心分配给每个子域。由于网状的拓扑性，块分区方案是最好的解决方案。在开始模拟时执行分解域的操作。子域的位置和大小在初始化后不会改变。系统向网络中的每个核心提供自己的网格的行和列的索引，以及每行和每列中存在的核心总数。因此，每个内核都可以直接计算出自己的子域的大小和原点。这种分区方案的作用是使得网状拓扑中的物理邻居的核心在相邻子域上运行。这对于优化整体减小延迟非常重要，因为大量数据依赖于相邻的子域。

为了更有效率，流体域的数据被放在一个数组中。这样可以在算法的不同阶段中最大化空间和时间的利用，并且能够降低由于存储器层次结构中的停滞而引起的性能下降。

2.2.1 增加源

对于速度场和密度场增加力是一种简单平行的操作，不需要在子域边界之间进行任何数据的操作，考虑 F 作为外力的向量场，S 是密度源的外标量场，公式可以表示如下：

速度公式为：$U_{i,j}^{n+1} = U_{i,j}^{n} + dtF_{i,j}$　密度公式为：$D_{i,j}^{n+1} = D_{i,j}^{n} + dtS_{i,j}$

代码：

```
void add_density(int px, int py, int r = 4, float value = 0.5f) {
    for (int y = -r; y <= r; y++) for (int x = -r; x <= r; x++) {
        float d = sqrtf(x*x + y*y);
        float u = smoothstep(float(r), 0.0f, d);
        old_density(px + x, py + y) += u*value;//高斯优化，边缘密度低，保证球形火种。
                                               //r:决定粒子个数！
    }
}
```

2.2.2 增加扩散

扩散阶段计算扩散对粒子网格中速度和密度的影响。h 表示粒子的大小。ν 表示流体粘度常数，κ 表示密度扩散常数。公式可以表示如下：

速度公式为：$U_{i,j}^{n+1} - \nu dt \frac{(U_{i-1,j}^{n+1} + U_{i+1,j}^{n+1} + U_{i,j-1}^{n+1} + U_{i,j+1}^{n+1} - 4U_{i,j}^{n+1})}{h^2} = U_{i,j}^{n}$

密度公式为：$D_{i,j}^{n+1} - \kappa dt \frac{(D_{i-1,j}^{n+1} + D_{i+1,j}^{n+1} + D_{i,j-1}^{n+1} + D_{i,j+1}^{n+1} - 4D_{i,j}^{n+1})}{h^2} = D_{i,j}^{n}$

代码：

```
void diffuse_density() {
    float diffusion = dt*100.0f;
    FOR_EACH_CELL{
       float diff = diffusion*(old_density(x - 1, y) + old_density(x + 1, y) + old_density(x,
y - 1) + old_density(x, y + 1));
    float sum = diff + old_density(x, y);
    new_density(x, y) = 1.0f / (1.0f + 4.0f*diffusion) * sum;
    }//5 个点的共同影响下的密度
    old_density.swap(new_density);
}

void diffuse_velocity() {
    float viscosity = dt*0.000001f;
    FOR_EACH_CELL{
        vec2f visc = viscosity*(old_velocity(x - 1, y) + old_velocity(x + 1, y) +
old_velocity(x, y - 1) + old_velocity(x, y + 1));
    vec2f sum = visc + old_velocity(x, y);
    new_velocity(x, y) = 1.0f / (1.0f + 4.0f*viscosity) * sum;
    }//5 个点的共同影响下的速度
    old_velocity.swap(new_velocity);
```

```
}
```

2.2.3 增加平流

平流阶段的目的是沿速度场移动密度和速度。h 表示粒子的大小， Δt 表示时间步长，Interp 表示 2D 线性插值函数。公式可以表示如下：

速度公式为： $U_{i,j}^{n+1} = Interp(U_{i,j}^{n}, \begin{pmatrix} i \\ j \end{pmatrix} - \frac{\Delta t}{h} U_{i,j}^{n})$

密度公式为： $D_{i,j}^{n+1} = Interp(D_{i,j}^{n}, \begin{pmatrix} i \\ j \end{pmatrix} - \frac{\Delta t}{h} U_{i,j}^{n})$

代码：

```
template <typename T>
T interpolate(const Grid<T> &grid, vec2f p) {//双线性插值
    int ix = floorf(p.x);
    int iy = floorf(p.y);
    float ux = p.x - ix;
    float uy = p.y - iy;
    return lerp(
        lerp(grid(ix + 0, iy + 0), grid(ix + 1, iy + 0), ux),
        lerp(grid(ix + 0, iy + 1), grid(ix + 1, iy + 1), ux),
        uy
    );
}



void advect_density() {
    FOR_EACH_CELL{
        vec2f pos = vec2f{ float(x),float(y) } - dt*old_velocity(x, y);
    new_density(x, y) = interpolate(old_density, pos);
    }//根据速度、dt 计算每个粒子的新位置，新的密度是原密度与位置的插值
    old_density.swap(new_density);
}
void advect_velocity() {
    FOR_EACH_CELL{
        vec2f pos = vec2f{ float(x),float(y) } - dt*old_velocity(x, y);
    new_velocity(x, y) = interpolate(old_velocity, pos);
    }//根据速度、dt 计算每个粒子的新位置,新的速度是原速度与位置的插值
    old_velocity.swap(new_velocity);
}
```

2.2.4 增加投影

投影阶段校正速度场以确保质量守恒，使流入和流出每个粒子相当。使用当前速度场获得当前的流动梯度场，P 表示泊松方程中的压力场。公式可以表示如下：

$$P_{i+1,j}+P_{i-1,j}+P_{i,j+1}+P_{i,j-1}-4P_{i,j}=(U^x_{i+1,j}-U^x_{i-1,j}+U^y_{i,j+1}-U^y_{i,j-1})h$$

代码：

```
void project_velocity() {
    Grid<float> p(nx, ny);
    Grid<float> p2(nx, ny);
    Grid<float> div(nx, ny);

    FOR_EACH_CELL{
        float dx = old_velocity(x + 1, y).x - old_velocity(x - 1, y).x;
        float dy = old_velocity(x, y + 1).y - old_velocity(x, y - 1).y;
        div(x, y) = dx + dy;//梯度
        p(x, y) = 0.0f;//压强场
    }

    for (int k = 0; k < iterations; k++) {
        FOR_EACH_CELL{
            float sum = -div(x, y) + p(x + 1, y) + p(x - 1, y) + p(x, y + 1) + p(x, y -
1);
        p2(x, y) = 0.25f*sum;
        }
        p.swap(p2);
    }//根据泊松定理，梯度计算以及周围的四个点来保证质量守恒，在这里 p 代表质量

    FOR_EACH_CELL{
        old_velocity(x, y).x -= 0.5f*(p(x + 1, y) - p(x - 1, y));
    old_velocity(x, y).y -= 0.5f*(p(x, y + 1) - p(x, y - 1));
    }
}
```

2.2.5 增加涡旋

涡旋现象，对于每个粒子计算涡流方向，计算得到新的速度。dir 为涡流方向，C 为涡线。公式可以表示如下：

速度公式为：$U^{n+1}_{i,j}=U^n_{i,j}+\Delta t*C_{i,j}*dir$

代码：

```
void vorticity_confinement() {//加入涡量的影响，简化版的涡量计算
    Grid<float> abs_curl(nx, ny);

    FOR_EACH_CELL{
        abs_curl(x, y) = fabsf(curl(x, y));
    }//首先计算每个点的涡线

        FOR_EACH_CELL{
```

```
        vec2f direction;
    direction.x = abs_curl(x, y - 1) - abs_curl(x, y + 1);
    direction.y = abs_curl(x + 1, y) - abs_curl(x - 1, y);
    //printvec(direction);
    direction = vorticity / ( sqrtf(dot(direction, direction)) + 1e-5f) * direction;
    //printvec(direction);
    if (!(x > nx*0.85f && !ray)) direction *= 0.0f;
    new_velocity(x, y) = old_velocity(x, y) + dt*curl(x, y)*direction;
    }//通过涡线计算每个点的涡流方向，然后原速度加上时间乘以涡线以及方向，得到新的速度
    old_velocity.swap(new_velocity);
}
```

三、演示操作

a：向左吹的风

d：向右吹的风

+：增大火焰

-：减小火焰

enter：还原

space：背景

发生器：稳定火焰，炸弹：涡旋火焰，地球：稳定液体

具体效果请参考视频。

参考资料：

[1] J. Stam, “Stable fluids”, In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, pp.121-128

[2] J. Stam, “Real-time fluid dynamics for games”, Proceedings of the Game Developer Corner, March 2003

[3] Fais M, Iorio F. Fast Fluid Dynamics on the Single-chip Cloud Computer[C]// Many-Core Applications Research Community. DBLP, 2013:59-63.