HW3

111550143 林彥佑

- Implementation

particleRelaxation():

particleRelaxation()函式負責實作流體模擬中的粒子鬆弛處理，以避免粒子之間的重疊。

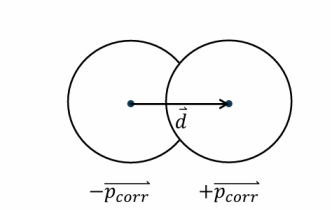
```
for (int i = 0; i < relaxation_cell_rows; ++i) {
    for (int j = 0; j < relaxation_cell_cols; ++j) {
        relaxation_cell_particle_ids[i][j].clear();
    }
}

for (int i = 0; i<num_particles; i++)
{
    int cell_x = static_cast<int>(particle_pos[i].x() / relaxation_cell_dim);
    int cell_y = static_cast<int>(particle_pos[i].y() / relaxation_cell_dim);

    cell_x = std::max(0, std::min(cell_x, relaxation_cell_cols - 1));
    cell_y = std::max(0, std::min(cell_y, relaxation_cell_rows - 1));

    relaxation_cell_particle_ids[cell_y][cell_x].push_back(i);
}
```

根據每個粒子的位置，計算其所屬的 relaxation cell，並將粒子 index 放入對應的 cell 中。

```
for (int iter = 0; iter < iterations; ++iter)
{
    for (int i = 0; i < relaxation_cell_rows; ++i) for (int j = 0; j < relaxation_cell_cols; ++j)
    {
        // TODO: Perform particle relaxation
        for (int p1 : relaxation_cell_particle_ids[i][j])
        {
            for (int k = -1; k <= 1; ++k) for (int l = -1; l <= 1; ++l)
            {
                int neighbor_cell_x = j + k;
                int neighbor_cell_y = i + l;

                if (neighbor_cell_x < 0 || neighbor_cell_x >= relaxation_cell_cols ||
                    neighbor_cell_y < 0 || neighbor_cell_y >= relaxation_cell_rows)
                    continue;

                for (int p2 : relaxation_cell_particle_ids[neighbor_cell_y][neighbor_cell_x])
                {
                    if (p1 == p2) continue;

                    float d = (particle_pos[p1] - particle_pos[p2]).norm();
                    if (d < 2 * particle_radius)
                    {
                        Eigen::Vector2f diff = particle_pos[p1] - particle_pos[p2];
                        float dist = diff.norm();
                        if (dist <= 1e-6f)
                            continue;
                        diff /= dist;
                        particle_pos[p1] += diff * (2 * particle_radius - dist) / 2.0f;
                        particle_pos[p2] -= diff * (2 * particle_radius - dist) / 2.0f;
                    }
                }
            }
        }
    }
```

進行 iterations 次的疊代，對於每個 relaxation cell 中的粒子，掃描該 cell 周圍 3x3 的鄰近 cell，以找出可能與之接觸的粒子，檢查粒子間距是否小於 2 * particle_radius，若是則視為碰撞。

使用公式



$$\overrightarrow{p_{corr}} = \frac{1}{2} * \frac{2r - |\vec{d}|}{|\vec{d}|} * \vec{d}$$

將兩粒子沿其連線方向等量推開。

transferVelocities(bool to_cell):

這個函式實作了 PIC/FLIP 模擬中粒子與 MAC 格點速度場之間的雙向速度轉換，根據 to_cell 執行 particle to cell 或 cell to particle 的轉換邏輯。

```
const float l = cell_dim;
std::vector<std::vector<Eigen::Vector2f>> weight_sum(cell_rows, std::vector<Eigen::Vector2f>(cell_cols, Eigen::Vector2f::Zero()));

if (to_cell)
{
    // TODO: Reset velocities and update cell types
    for (int i = 0; i < cell_rows; ++i) for (int j = 0; j < cell_cols; ++j)
    {
        cell_velocities[i][j] = Eigen::Vector2f::Zero();
        weight_sum[i][j] = Eigen::Vector2f::Zero();
        if (cell_types[i][j] != CellType::SOLID)
            cell_types[i][j] = CellType::AIR;
    }

    for (int i = 0; i < num_particles; ++i)
    {
        Eigen::Vector2f& pos = particle_pos[i];
        int j0 = (floor(pos.x() / l));
        int i0 = (floor(pos.y() / l));
        j0 = std::max(0, std::min(j0, cell_cols - 1));
        i0 = std::max(0, std::min(i0, cell_rows - 1));

        if (cell_types[i0][j0] != CellType::SOLID) cell_types[i0][j0] = CellType::FLUID;
    }
}
```

首先，reset 所有 cell 速度以及 cell types，不是 SOLID 先設為 AIR，再判斷該 cell 是否有粒子，設為 FLUID。

```
for (int component = 0; component < 2; ++component)
{
    // Might need some setup here

    for (int i = 0; i < num_particles; ++i)
    {
        // TODO: Bilinear interpolation on staggered grid
        Eigen::Vector2f& pos = particle_pos[i];
        Eigen::Vector2f& vel = particle_vel[i];
        float x = pos.x();
        float y = pos.y();

        if (component == 0)
        {
            y -= (0.5f * l);
        }
        else
        {
            x -= (0.5f * l);
        }

        int j0 = (floor(x / l));
        int i0 = (floor(y / l));
        j0 = std::max(0, std::min(j0, cell_cols - 2));
        i0 = std::max(0, std::min(i0, cell_rows - 2));
        int j1 = j0 + 1;
        int i1 = i0 + 1;

        float deltax = x / l - j0;
        float deltay = y / l - i0;
        float w1 = (1 - deltax) * (1 - deltay);
        float w2 = (deltax) * (1 - deltay);
        float w3 = (deltax) * (deltay);
        float w4 = (1 - deltax) * (deltay);
```

```cpp
float v = (component == 0) ? vel.x() : vel.y();

if (to_cell)
{
    // TODO: Transfer particle velocities to cells and store weights
    cell_velocities[i0][j0][component] += w1 * v;
    cell_velocities[i0][j1][component] += w2 * v;
    cell_velocities[i1][j0][component] += w4 * v;
    cell_velocities[i1][j1][component] += w3 * v;

    weight_sum[i0][j0][component] += w1;
    weight_sum[i0][j1][component] += w2;
    weight_sum[i1][j0][component] += w4;
    weight_sum[i1][j1][component] += w3;
}
else
{
    // TODO: Transfer valid cell velocities back to the particles using a mixture of PIC and FLIP
    std::vector<std::pair<int, int>> idx = { {i0, j0}, {i0, j1}, {i1, j0}, {i1, j1} };
    std::vector<float> weights = { w1, w2, w4, w3 };
    float w_sum = 0.0f, pic_sum = 0.0f, flip_sum = 0.0f;
```

```cpp
    for (int k = 0; k < 4; ++k)
    {
        int i = idx[k].first;
        int j = idx[k].second;
        bool isValid = false;
        if (i - 1 < 0 || j - 1 < 0) {
            isValid = false;
        }
        else if (component == 0) {
            isValid = (cell_types[i][j] == FLUID || cell_types[i][j - 1] == FLUID);
        }
        else {
            isValid = (cell_types[i][j] == FLUID || cell_types[i - 1][j] == FLUID);
        }

        if (isValid)
        {
            float w = weights[k];
            w_sum += w;
            pic_sum += w * cell_velocities[i][j][component];
            flip_sum += w * (cell_velocities[i][j][component] - prev_cell_velocities[i][j][component]);
        }
    }

    if (w_sum > 0.0f)
    {
        float pic = pic_sum / w_sum;
        float flip = particle_vel[i][component] + flip_sum / w_sum;
        particle_vel[i][component] = (1 - flip_ratio) * pic + flip_ratio * flip;
    }
}
```

```cpp
if (to_cell)
{
    // TODO: Normalize cell velocities and store a backup in prev_velocities.
    for (int i = 0; i < cell_rows; ++i) for (int j = 0; j < cell_cols; ++j)
    {
        if (weight_sum[i][j][component] > 0.0f)
        {
            cell_velocities[i][j][component] /= weight_sum[i][j][component];
            prev_cell_velocities[i][j][component] = cell_velocities[i][j][component];
        }
        else
        {
            prev_cell_velocities[i][j][component] = cell_velocities[i][j][component];
        }
    }
}
```

calculate the 4 cells that should contribute to the biliear interpolation of each particle. 照講義公式，先-l/2 得出最左下角的(I, j)，得出 i0, j0, i1, j1，然後計算 deltax, deltay，我有點被講義給的公式誤導，我原本是分成 x 和 sample_x 去計算，x 沒有也減去 l/2，導致粒子亂飛，再來如果照講義公式，sample_x 會被除以兩次 l，這是完全錯誤的，我不知道我是否誤解講義的意思，講義上是否在算 floor(sample_x/l)後要*l，雖說影片有講到比較正確的算法，但講義還是有混淆到我。得出 delta 之後套公式

$$\Delta x = x - \lfloor \frac{x_{sample}}{l} \rfloor \quad \Delta y = y - \lfloor \frac{y_{sample}}{l} \rfloor$$

$$w_1 = \left(1 - \frac{\Delta x}{l}\right)\left(1 - \frac{\Delta y}{l}\right)$$

$$w_2 = \left(\frac{\Delta x}{l}\right)\left(1 - \frac{\Delta y}{l}\right)$$

$$w_3 = \left(\frac{\Delta x}{l}\right)\left(\frac{\Delta y}{l}\right)$$

$$w_4 = \left(1 - \frac{\Delta x}{l}\right)\left(\frac{\Delta y}{l}\right)$$

得出 w1~w4，將該粒子的速度分別加權加總到這四個格點的對應速度分量上（x 或 y），並在 weight_sum 中記錄該格點累積了多少權重，最後將格點的速度除以其累積權重，完成加權平均，並同時備份至 prev_cell_velocities 供下一步 FLIP 使用。

to_cell=false，將格點上的速度重新插值還原回粒子，使粒子能夠接續進行運動，對於每個粒子，我們同樣根據其位置與格點進行 biliear interpolation，要排除兩 cell 皆為 AIR 的速度干擾，用 isValid 去判斷是否採用速度，使用有效的格點速度計算

PIC 速度：直接插值得到的新速度。

FLIP 速度：透過 cell_velocities - prev_cell_velocities 取得的差值，加回粒子原速度。

最終將 PIC 與 FLIP 結果依據 flip_ratio 混合，更新回粒子速度。

$$v_{pic\,x} = \frac{w_1 v_{x1} + w_2 v_{x2} + w_4 v_{x4}}{w_1 + w_2 + w_4}$$

$$v_{flip\,x} = v_{px} + \frac{w_1(v_{x1} - v_{xprev1}) + w_2(v_{x2} - v_{xprev4}) + w_4(v_{x4} - v_{xprev4})}{w_1 + w_2 + w_4}$$

$$v_{px} = \alpha_{flip} v_{flipx} + (1 - \alpha_{flip}) v_{picx}$$

updateDensity():

updateDensity()的目的是計算每個格點的密度（cell density），作為下一步 solveIncompressibility()判斷是否需要壓縮修正的依據。

```
for (int i = 0; i < cell_rows; ++i) for (int j = 0; j < cell_cols; ++j)
    cell_densities[i][j] = 0.0f;

const float l = cell_dim;

for (int i = 0; i < num_particles; ++i)
{
    // TODO: Perform bilinear interpolation
    Eigen::Vector2f& pos = particle_pos[i];
    float x = pos.x();
    float y = pos.y();
    x -= 0.5f * l;
    y -= 0.5f * l;

    int j0 = (floor(x / l));
    int i0 = (floor(y / l));
    j0 = std::max(0, std::min(j0, cell_cols - 2));
    i0 = std::max(0, std::min(i0, cell_rows - 2));
    int j1 = j0 + 1;
    int i1 = i0 + 1;

    float deltax = x / l - j0;
    float deltay = y / l - i0;
    float w1 = (1 - deltax) * (1 - deltay);
    float w2 = (deltax) * (1 - deltay);
    float w3 = (deltax) * (deltay);
    float w4 = (1 - deltax) * (deltay);
    cell_densities[i0][j0] += w1;
    cell_densities[i0][j1] += w2;
    cell_densities[i1][j1] += w3;
    cell_densities[i1][j0] += w4;
}
```

```
if (particle_rest_density == 0.0f)
{
    // TODO: Calculate resting particle densities in fluid cells.
    float density_sum = 0.0f;
    int num_fluid_cells = 0;
    for (int i = 0; i < cell_rows; ++i) for (int j = 0; j < cell_cols; ++j)
    {
        if (cell_types[i][j] == CellType::FLUID)
        {
            density_sum += cell_densities[i][j];
            num_fluid_cells++;
        }
    }

    if (num_fluid_cells > 0)
        particle_rest_density = density_sum / num_fluid_cells;
}
```

一開始將所有格點的密度 cell_densities[i][j]歸零，對每個粒子先將其位置調整為落在格點中心座標（即 x -= 0.5 * cell_dim, y -= 0.5 * cell_dim），接著，根據粒子位置找到其對應的上下左右四格，利用 biliear interpolation 計算四個權重 w1~w4，並將每個粒子對應的權重加總至四個格點的密度，僅在 particle_rest_density == 0.0 時計算 rest_density，掃描所有格點，若該格點型態為 FLUID，則累加其密度值並統計數量。

$$\rho_{rest} = \frac{Sum\ of\ fluid\ cell\ densities}{N_{fluid\ cells}}$$

solveIncompressibility():

solveIncompressibility()的目的是讓整個流體系統滿足不可壓縮性條件，也就是確保每個流體格子的流入與流出速率相等。

```
const float l = cell_dim;

for (int iter = 0; iter < iterations; ++iter)
{
    for (int i = 1; i < cell_rows - 1; ++i) for (int j = 1; j < cell_cols - 1; ++j)
    {
        if (cell_types[i][j] != CellType::FLUID)
            continue;
        // TODO: Calculate divergence of fluid cells

        // TODO: Add bias to ouflow if density_correction is true

        // TODO: Correct the cell velocities
        bool left = (cell_types[i][j - 1] != CellType::SOLID);
        bool right = (cell_types[i][j + 1] != CellType::SOLID);
        bool up = (cell_types[i + 1][j] != CellType::SOLID);
        bool down = (cell_types[i - 1][j] != CellType::SOLID);

        int num_neighbors = int(left) + int(right) + int(up) + int(down);
        if (num_neighbors == 0)
            continue;

        float vx_r = right ? cell_velocities[i][j + 1].x() : 0.0f;
        float vx_l = left  ? cell_velocities[i][j].x() : 0.0f;
        float vy_u = up    ? cell_velocities[i + 1][j].y() : 0.0f;
        float vy_d = down  ? cell_velocities[i][j].y() : 0.0f;
        float divergence = vx_r - vx_l + vy_u - vy_d;
```

```
if (density_correction)
{
    float density_diff = cell_densities[i][j] - particle_rest_density;
    if (density_diff > 0.0f)
        divergence -= stiffness_coefficient * density_diff;
}

float correction = divergence / num_neighbors * over_relaxation;

if (left)
    cell_velocities[i][j].x() += correction;
if (right)
    cell_velocities[i][j + 1].x() -= correction;
if (up)
    cell_velocities[i + 1][j].y() -= correction;
if (down)
    cell_velocities[i][j].y() += correction;
```

外層使用 iterations 迴圈對整體流場進行多次修正，使每個格子的 divergence 趨近 0，內層針對

所有 fluid 格子逐一處理，要判斷該格的鄰居是否是 SOLID，計算上下左右格非 SOLID 個數，並得出上下左右速度，套過公式:

$$div = v_{top} + v_{right} - v_{left} - v_{bottom}$$

得出 divergence，若 density_correction == true，額外加入修正項

$$div -= k_{stiff}(\rho_{cell} - \rho_{rest})$$

修正量會根據目前 divergence、鄰近可流通的格子數（上下左右非 solid）與 over_relaxation 係數決定

$$v_{corr} = o * \frac{div}{dirs}$$

左/下方向的速度加上 correction，右/上方向的速度減去 correction

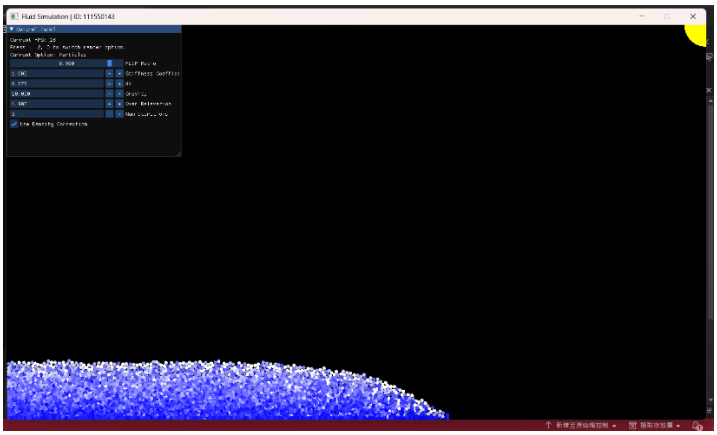
- Results & Discussion

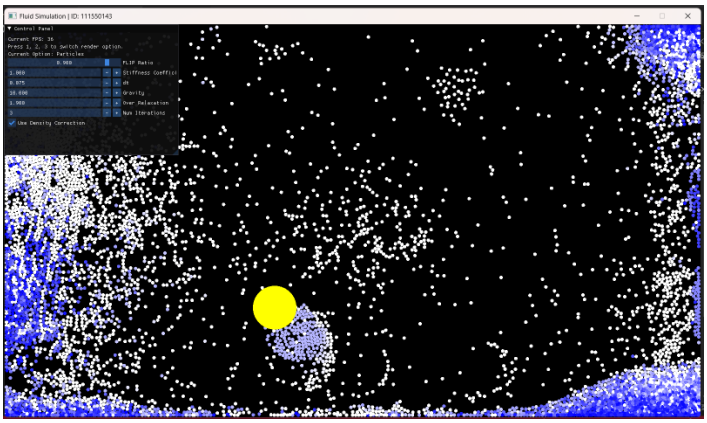
cell_dim:

cell_dim=10:

正常，如 demo 影片所示。

cell_dim=1:

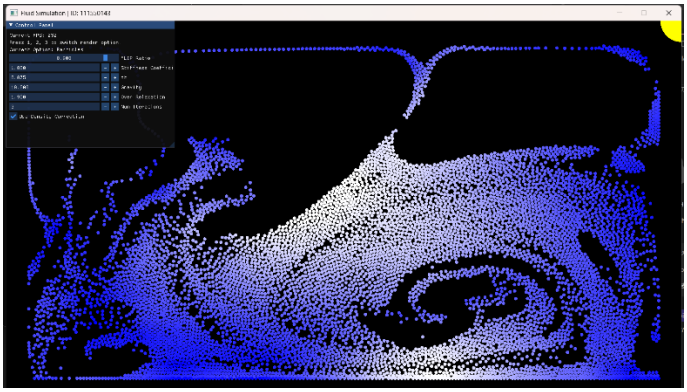


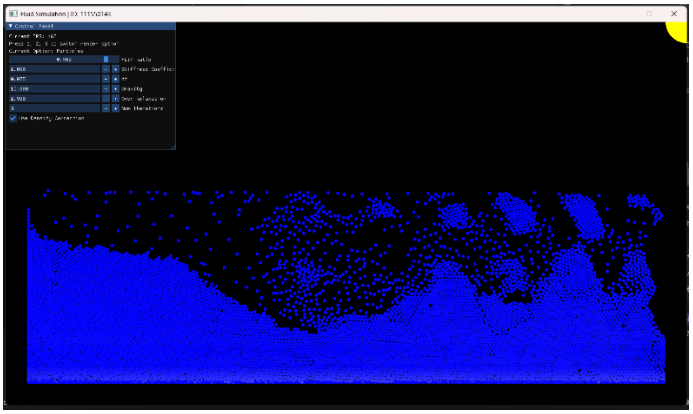


每格只包含少量粒子，造成每格的密度估計較不穩定（浮動大）。在 FLIP 模型中，粒子速度從格點插值回來的品質變差。細節多、可以模擬出更精細的流體動態，但初始流速慢。

cell_dim=50

每格涵蓋大量粒子，導致密度估計過於模糊。粒子的變化趨於平均化，流體行為顯得「鈍化」。流動區域會看起來像硬塊（因為速度修正範圍變少）。





FLIP ratio:

當 flip_ratio = 0.0，完全採用 PIC 方法。

當 flip_ratio = 1.0，完全採用 FLIP 方法。

flip_ratio = 0.0

粒子速度每次都被格點速度完全取代，模擬非常穩定，不容易跳動或爆炸，多粒子一起行動。穩定、不抖動，水波紋或渦流會快速消散。

flip_ratio = 1.0

粒子速度每次都加上從格點推回的速度變化量（delta）。

能量與動量保留較完整，流動細節更自然，但容易因誤差累積導致粒子亂飛，尤其在格點密度變化大或靠近邊界時，數值震盪會放大，粒子會出現單獨一顆飛的現象。

保留細節、模擬活潑、易抖動、不穩定

Stiffness coefficient:

stiffness_coefficient = 0.0

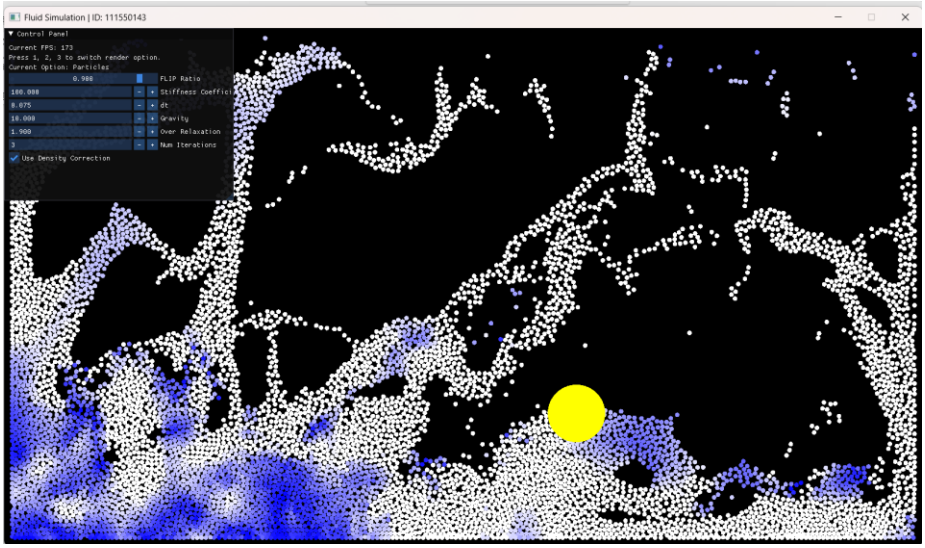
若粒子因邊界或碰撞集中於一格，則該格密度會異常高，無機制將其膨脹釋放，速度場穩定。

stiffness_coefficient = 1.0

正常

stiffness_coefficient = 100.0

密度只要略高於靜態密度就會被強力推開，模擬畫面中會看到粒子不斷抖動，像爆炸一樣四散。數值震盪、易發散、不穩定


Fluid Simulation | ID: 111550143
Control Panel
Current FPS: 173
Press 1, 2, 3 to switch render option.
Current Option: Particles
0.988
FLIP Ratio
100.000
Stiffness Coefficie
0.075
dt
10.000
Gravity
1.900
Over Relaxation
3
Num Iterations
Use Density Correction