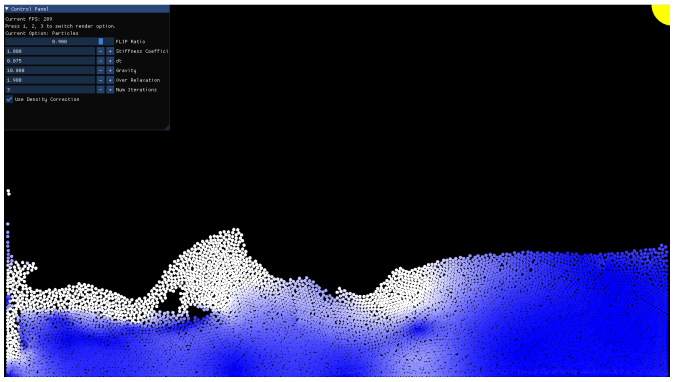
Computer Animation & Special Effects: HW3 Report

Author: 111550113 謝詠晴

1 Introduction

此次作業旨在實作基於 PIC (Particle-In-Cell) 與 FLIP (Fluid-Implicit Particle) 方法的 2D 流體模擬系統，使用粒子來代表流體的運動，並透過網格來記錄速度場的更新，此外，透過流體動力學中的質量守恆原理，求解不可壓縮條件，並透過壓力修正來校正速度場，確保流體的體積保持穩定。



2 Implementation

2.1 particleRelaxation

- 目的:

  處理粒子之間的碰撞並進行位置調整，以確保粒子間距離至少是 2 * particle_radius 而

  不會互相重疊。

- 實作方法:

  (1) 根據每個粒子的位置 (x,y)，透過除以 cell 邊長的方式，計算它映射到哪一格 relaxation cell

```
for(int i = 0; i < num_particles; ++i) {
    Eigen::Vector2f& pos = particle_pos[i];
    int cell_x = static_cast<int>(pos.x() / relaxation_cell_dim);
    int cell_y = static_cast<int>(pos.y() / relaxation_cell_dim);

    if (cell_x >= 0 && cell_x < relaxation_cell_cols && cell_y >= 0 && cell_y < relaxation_cell_rows)
        relaxation_cell_particle_ids[cell_y][cell_x].push_back(i);
}
```

  (2) 外層的 iteration 代表修正次數，確保粒子達到穩定狀態

  (3) 內層的迴圈則遍歷每格 cell，以及用相對位置 -1~1 去遍歷他的 3x3 相鄰區域

  (4) 我會先檢查是否超出邊界，p1 代表目前處理的 cell 中的粒子，p2 則是鄰居 cell 中的粒子，為了避免重複檢查，用 if (p1 >= p2) continue; 排除無效比較

```
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 x = -1; x <= 1; ++x) for(int y = -1; y <= 1; ++y)
        {
            int cell_row = i + y; // row
            int cell_col = j + x; // col

            if (cell_col >= 0 && cell_col < relaxation_cell_cols && cell_row >= 0 && cell_row < relaxation_cell_rows){
                const auto& neighbor_particles = relaxation_cell_particle_ids[cell_row][cell_col];

                for (int p1 : relaxation_cell_particle_ids[i][j]) {
                    for (int p2 : relaxation_cell_particle_ids[cell_row][cell_col]) {
                        if (p1 >= p2) continue;
```

(5) 計算粒子間的距離，若小於2 * particle_radius，則計算要修正的距離進行位置的調整。避免計算 direction 時除以零，因此我將 dist 限制在最小 1e-6f

```
Eigen::Vector2f d = particle_pos[p1] - particle_pos[p2];
float dist = std::max(d.norm(), 1e-6f);
if (dist < 2 * particle_radius) {
    Eigen::Vector2f direction = d / dist;
    Eigen::Vector2f correction = 0.5f * direction * (2 * particle_radius - dist);
    particle_pos[p1] += correction;
    particle_pos[p2] -= correction;
```

2.2 transferVelocities

- 目的：
  粒子和網格間傳遞速度資訊，當 to_cell = true; 將粒子的速度轉換並更新到網格；當 to_cell = false：從網格反向更新粒子的速度。
- [粒子->網格] 實作方法：
  (1) 初始化將所有格子的速度設為零
  (2) 遍歷所有粒子，根據粒子位置計算他所在的格子位置，並將此格的 CellType 設為 FLUID

```
// assign cell types!!
for (int i = 0; i < num_particles; ++i)
{
    // Eigen::Vector2f& pos = particle_pos[i];
    int cell_x = static_cast<int>(std::floor(particle_pos[i].x() / cell_dim));
    int cell_y = static_cast<int>(std::floor(particle_pos[i].y() / cell_dim));

    if (cell_x >= 0 && cell_x < cell_cols && cell_y >= 0 && cell_y < cell_rows) {
        if (cell_types[cell_y][cell_x] != CellType::SOLID) {
            cell_types[cell_y][cell_x] = CellType::FLUID;
        }
    }
}
```

(3) Bilinear Interpolation：先將粒子的位置轉換到 MAC 網格座標，並根據 x 或 y component，對 base_y 或 base_x 減去 0.5 來對齊格子的左邊界或下邊界。透過 std::floor() 來取得網格的整數座標，計算粒子在網格座標內部的相對位置 dx 和 dy，並計算 Bilinear Interpolation 對網格四個頂點的影響權重 w1~w4。

```
// TODO: Bilinear interpolation on staggered grid
Eigen::Vector2f& pos = particle_pos[i];
Eigen::Vector2f& vel = particle_vel[i];

float base_x = pos.x() / cell_dim;
float base_y = pos.y() / cell_dim;
component == 0 ? base_y-=0.5f : base_x-=0.5f;
int cell_x = static_cast<int>(std::floor(base_x));
int cell_y = static_cast<int>(std::floor(base_y));
float dx = base_x - cell_x;
float dy = base_y - cell_y;

// weights
float w1 = (1 - dx) * (1 - dy);
float w2 = dx * (1 - dy);
float w3 = dx * dy;
float w4 = (1 - dx) * dy;

if(cell_x < 0 || cell_x + 1 >= cell_cols || cell_y < 0 || cell_y + 1>= cell_rows) continue;
```

(4) 接著將粒子的速度插值並累加到4個相鄰的網格 cell_velocities，並且會透過 weight_accumulator 先暫存累加的權重，以利後續正規化使用。x 和 y 分量會分開計算但步驟相同。

(5) 最後一樣針對 x 和 y 分量分開處理，將 cell_velocites 除以 weight_accumulator 進行速度正規化，並且會將目前的速度值也先 assign 給 prev_cell_velocities，以利 FLIP 計算使用。

```
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 (component == 0){
            if(weight_accumulator[i][j] > 0.0f){
                cell_velocities[i][j].x() /= weight_accumulator[i][j];
            }
            prev_cell_velocities[i][j].x() = cell_velocities[i][j].x();
        }
        else {
            if(weight_accumulator[i][j] > 0.0f){
                cell_velocities[i][j].y() /= weight_accumulator[i][j];
            }
            prev_cell_velocities[i][j].y() = cell_velocities[i][j].y();
        }
    }
}
```

- [網格->粒子] 實作方法：

(1) 一樣會分成 x 和 y 分量實作，判斷在計算 pic_velocity 或是 flip_velocity 是否要考慮此位置的條件：x 分量是判斷此格或左邊前一格是否至少有一格是 FLUID，y 分量則是改判斷此格或是下面一格。

```
// TODO: Transfer valid cell velocities back to the particles using a mixture of PIC and FLIP
Eigen::Vector2f pic_velocity = Eigen::Vector2f::Zero();
Eigen::Vector2f flip_velocity = Eigen::Vector2f::Zero();

if (component == 0) { // x component
    float weight_sum_x = 0.0f;
    float vx1 = 0.0f, vx2 = 0.0f, vx3 = 0.0f, vx4 = 0.0f;
    float vx1_prev = 0.0f, vx2_prev = 0.0f, vx3_prev = 0.0f, vx4_prev = 0.0f;

    if ((cell_types[cell_y][cell_x] == CellType::FLUID) ||
        (cell_x > 0 && cell_types[cell_y][cell_x - 1] == CellType::FLUID)) {
        vx1 = cell_velocities[cell_y][cell_x].x();
        vx1_prev = prev_cell_velocities[cell_y][cell_x].x();
        weight_sum_x += w1;
    }
```

(2) 最後依照 PIC 和 FLIP 計算公式完成，因為vx 和 vx_prev 都會先初始化為0，這樣若判斷條件不符，兩者都是0，計算時就不會計算到此位置。

```
// PIC
if (weight_sum_x > 0.0f) {
    pic_velocity.x() = (w1 * vx1 + w2 * vx2 + w3 * vx3 + w4 * vx4) / weight_sum_x;
}
// FLIP
 if (weight_sum_x > 0.0f) {
    flip_velocity.x() = vel.x() +
        (w1 * (vx1 - vx1_prev) + w2 * (vx2 - vx2_prev) + w3 * (vx3 - vx3_prev) + w4 * (vx4 - vx4_prev)
}
// TOTAL
vel.x() = (1 - flip_ratio) * pic_velocity.x() + flip_ratio * flip_velocity.x();
```

2.3 updateDensity

- 目的:

  計算每個網格的密度和流體粒子靜態時的密度，這些資訊會在下一步檢查網格是否過度壓縮而進行的壓力修正使用到。

- 實作方法:

  (1) 遍歷每個粒子，計算他在網格的位置，這邊 x 和 y 都需要減去 0.5 對齊到網格中心，和前一個 function 一樣的方法計算雙線性插值的權重，對該粒子所在的四個相鄰網格的密度加上 w1~w4 權重。

```
if (cell_x >= 0 && cell_x + 1 < cell_cols && cell_y >= 0 && cell_y + 1 < cell_rows) {
    cell_densities[cell_y][cell_x] += w1;
    cell_densities[cell_y][cell_x + 1] += w2;
    cell_densities[cell_y + 1][cell_x + 1] += w3;
    cell_densities[cell_y + 1][cell_x] += w4;
}
```

(2) 計算靜態密度，遍歷所有的 cell，若該格 cell_type 是 FLUID，則會累加他的密度到 total_density，num_fluid_cells 也會加一，最後計算所有流體 cell 的平均。

```
if (particle_rest_density == 0.0)
{
    // TODO: Calculate resting particle densities in fluid cells.
    float total_density = 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) {
            total_density += cell_densities[i][j];
            num_fluid_cells++;
        }
    }
    if (num_fluid_cells > 0) {
        particle_rest_density = total_density / num_fluid_cells;
    }
}
```

2.4 solveIncompressibility

- 目的:

  強制流體維持不可壓縮性，修正流體單元的速度，使其流入與流出相等

- 實作方法:

  (1) 計算 divergence 時會考慮上下左右相鄰格子是否是 solid，若是就不會考慮該方向的 divergence，右和上方向是流出，左和下方向是流入

```
// TODO: Calculate divergence of fluid cells
if (cell_types[i][j] != CellType::FLUID) continue;
float divergence = 0.0f;
if (cell_types[i][j - 1] != CellType::SOLID) divergence -= cell_velocities[i][j].x();
if (cell_types[i][j + 1] != CellType::SOLID) divergence += cell_velocities[i][j+1].x();
if (cell_types[i - 1][j] != CellType::SOLID) divergence -= cell_velocities[i][j].y();
if (cell_types[i + 1][j] != CellType::SOLID) divergence += cell_velocities[i+1][j].y();
```

  (2) 密度修正:當該 cell 的密度大於靜態密度時，代表流體被過度壓縮，需要將這個偏差作為額外的 divergence 來源，用 stiffness_coefficient 來放大此影響，將 divergence 減去此偏差，使齊變更負，使流體因而想再擴張以修正。

```
// TODO: Add bias to ouflow if density_correction is true
if (density_correction) {
    float density_diff = cell_densities[i][j] - particle_rest_density;
    if (density_diff > 0.0f) {
        divergence -= stiffness_coefficient * density_diff;
    }
}
```

  (3) 速度修正:divergence > 0 表示流出太多，divergence < 0 表示流入太多。num_directions 計算有幾個方向可以流動，correction 會基於 divergence, over_relaxation, num_directions 計算修正量。當最後計算出的 divergence > 0，correction 的值就會正的，加到左和下方向的速度，使其流入更多來修正。

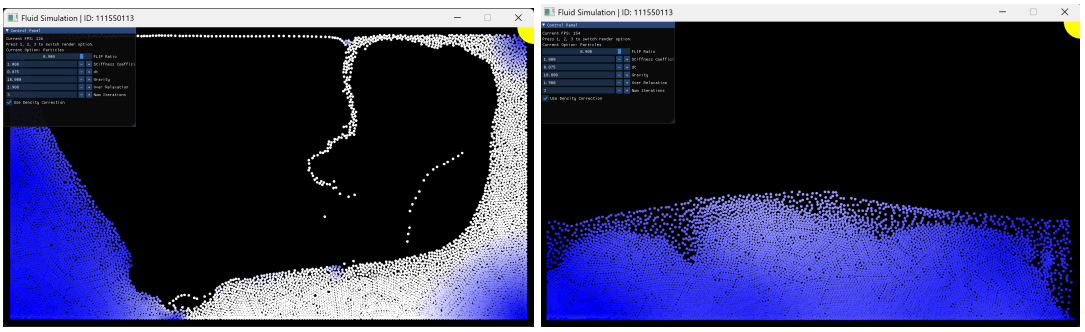
```cpp
// TODO: Correct the cell velocities
int num_directions = 0;
if (cell_types[i - 1][j] != CellType::SOLID) num_directions++;
if (cell_types[i + 1][j] != CellType::SOLID) num_directions++;
if (cell_types[i][j - 1] != CellType::SOLID) num_directions++;
if (cell_types[i][j + 1] != CellType::SOLID) num_directions++;
if (num_directions > 0) {
    float correction = over_relaxation * divergence / num_directions;
    if (cell_types[i][j - 1] != CellType::SOLID) cell_velocities[i][j].x() += correction;

    if (cell_types[i][j + 1] != CellType::SOLID) cell_velocities[i][j + 1].x() -= correction;

    if (cell_types[i - 1][j] != CellType::SOLID) cell_velocities[i][j].y() += correction;

    if (cell_types[i + 1][j] != CellType::SOLID) cell_velocities[i + 1][j].y() -= correction;
}
```

3 Results and Discussion

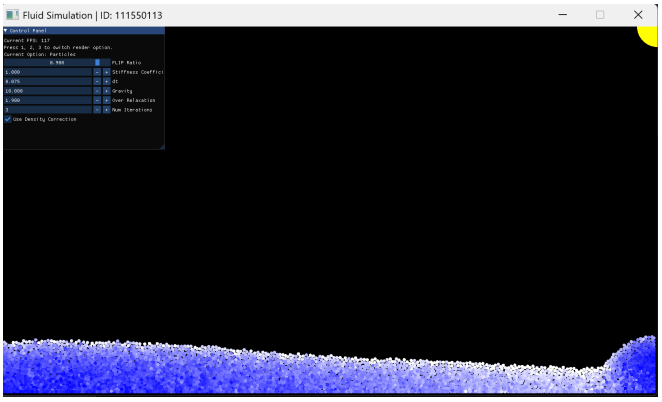
3.1 How different parameters affect the results

3.1.1 Cell dim

原本預設是10.0f，我將其調大為20.0f，粒子的分布會變得更為分散，粒子的間距也會更大，一開始流動就會濺的更高，靜態時可以明顯看到粒子間的有很大的距離，就不是自然的流體現象了。



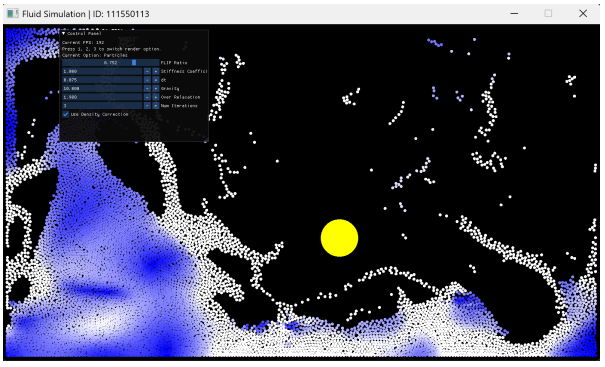
將 cell_dim 調小為5.0f，因為太擠了，使得粒子一開始墜落後沒有濺起的效果，會有粒子在原地快速攪動的現象。



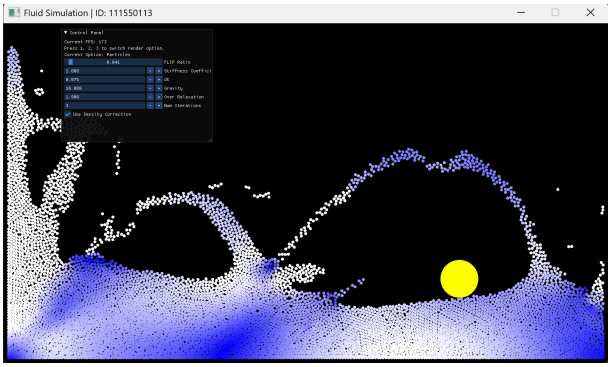
3.1.2 Flip ratio

flip_ratio 越小，粒子的黏著性越明顯，拿黃色圓球去攪動使粒子濺起時，flip_ratio 小濺起時會多個粒子黏在一起濺起，而 flip_ratio 大濺起時，則會變成單獨的多個粒子分開散開。

在攪動時的漩渦部分，高速旋轉下，flip_ratio 大時的表現較為真實，也可以更快更靈敏的根據黃色圓球的攪動改變，而 flip_ratio 小時渦流的模擬就較有阻尼感。



flip_ratio 大



flip_ratio 小

3.1.3 Stiffness coefficient

調整 stiffness_coefficient 的值從 0.1 ~ 2.5 的視覺效果整體來說不太明顯，要較細部觀察才看得出差別，當 stiffness 較高時，流體受到壓縮會更迅速的反彈回去，而當 stiffness 較低時，流體容易被壓縮，看起來的感覺較柔順，像是融化的液體般，形變較大。

3.1.4 Over Relaxation

over_relaxation 是用以修正 divergence，使其可以更快速趨近於 0，讓流體更快達到平衡。over_relaxation 較小時，粒子更密集和黏稠，拿黃色圓球干擾時，他的回彈速度慢；over_relaxation 越大時，粒子會越不緊密，回彈速度也較快。但太大時，會有白色粒子快速在不同位置閃動的現象，推測可能是數值震盪造成的不穩定顫動。

3.1.5 Num Iterations

num_iterations 越高表示修正次數越度，粒子的穩定度應該會較高。但增加 num_iterations 會使得計算過程變夠久，而導致粒子移動的速度變得更加緩慢，預設值為3，我調整為1或2時，粒子整體的移動速度就變快許多，而調整到 5 時就有感覺到粒子濺起後掉落的速度變慢了。

3.1.6 dt

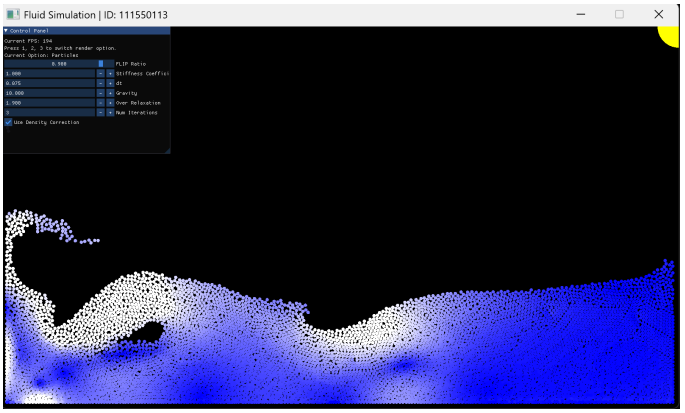
dt 越大，粒子的流動、滾動速度會越快，而 dt 越小則相反，粒子會像是較穩定的波浪。

3.1.7 Gravity

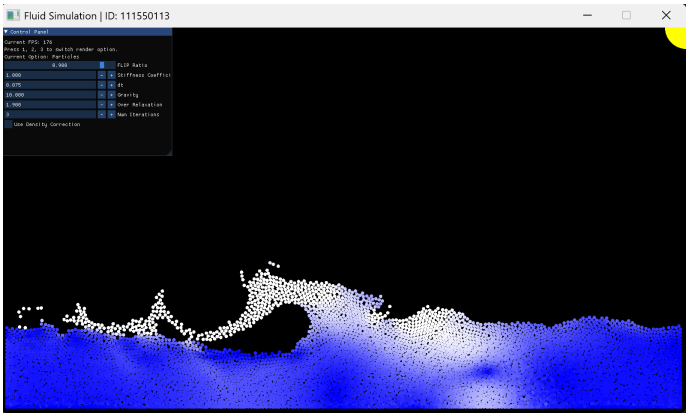
gravity 越大，粒子濺起後掉落下來的速度越快，當 gravity 設為 0 時，因為無重力，粒子濺起後會停留在天花板而無法掉落下來。

3.1.8 Density Correction

有開啟 density correction 時，粒子不會那麼緊密，因為受到擠壓時讓他能迅速反彈回去，更有水花四濺的效果，且擾動後也能更快回彈；而沒有開啟時會較沒有擴散感，更快就直接塌陷趨於平穩了，且會看起來有些部分被壓扁了。



with density_correction



without density_correction