Computer Animation Homework 3

112550184 陳璽安

1 Introduction

I implemented 2D fluid simulation in this assignment using the PIC/FLIP method. The components implemented include:

1. **Particle Relaxation** - Preventing particle overlap.
2. **Velocity Transfer** - Transfer particle velocity to cell or transfer from cell to particles.
3. **Density Calculation** - Computing fluid density for incompressibility.
4. **Solving Incompressibility** - Do divergence correction.

The implementation uses a staggered MAC grid for storing velocity and density to control simulation behavior.

2 Implementation Details

2.1 Particle Relaxation

For each cell, clear the relaxation cell list.

```
    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();
        }
    }
```

Iterate all particles, assign particles to cells based on their position.

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

        row = std::max(0, std::min(row, relaxation_cell_rows - 1));
        col = std::max(0, std::min(col, relaxation_cell_cols - 1));
        relaxation_cell_particle_ids[row][col].push_back(i);
    }
```

Perform relaxation in each iteration:

For each relaxation cell, iterate the particles in the relaxation particle list. And for each particle, iterate the particles in the cells that are adjacent to the particle's relaxation cell(include the cell of itself). If the distance between two particles is less than two times of particle radius, perform relaxation.

Compute the correction vector by $\vec{p}_{corr} = \frac{1}{2} \cdot \frac{2r - |\vec{d}|}{|\vec{d}|} \cdot \vec{d}$, and add or subtract the particles' positions to the correction vector.

```
    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 (size_t p = 0; p < relaxation_cell_particle_ids[i][j].size(); p++) {
                int pid = relaxation_cell_particle_ids[i][j][p];
                Eigen::Vector2f pos_p = particle_pos[pid];

```

```
                for (int ni = std::max(0, i - 1); ni <= std::min(relaxation_cell_rows - 1, i + 1); ni++) {
                    for (int nj = std::max(0, j - 1); nj <= std::min(relaxation_cell_cols - 1, j + 1);nj+
                    ↪  +) {
                        for (size_t q = 0; q < relaxation_cell_particle_ids[ni][nj].size(); q++) {
                            int qid = relaxation_cell_particle_ids[ni][nj][q];
                            if (pid == qid) continue;
                            Eigen::Vector2f pos_q = particle_pos[qid];
                            Eigen::Vector2f d = pos_p - pos_q;

                            float d2 = d.squaredNorm();
                            float min_d = 2.0f * particle_radius;
                            if (d2 < min_d * min_d&&d2>0.0f) {
                                float dist = std::sqrt(d2);
                                //(1/2) * (2r - |d|) * (d/|d|)
                                Eigen::RowVector2f corr = 0.5f * (min_d - dist) * (d / dist);

                                particle_pos[pid] += corr;
                                particle_pos[qid] -= corr;
                            }
                        }
                    }
                }
            }
        }
    }
```

2.2 Transfer Velocity

Reset velocity of each cell to be zero and update cell type.(Default to be air.)

```
    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();
            //default to be air
            if (cell_types[i][j] != CellType::SOLID)
                cell_types[i][j] = CellType::AIR;
        }
    }
```

Construct two arrays to store weight accumulations in x and y direction respectively.

```
    std::vector<std::vector<float>> weight_sum_x(cell_rows, std::vector<float>(cell_cols+1, 0.0f));
    std::vector<std::vector<float>> weight_sum_y(cell_rows, std::vector<float>(cell_cols+1, 0.0f));
```

Iterate over x and y component. (`component=0`: Vx, `component=1`: Vy). Calculates the sample positions (gx, gy) of each velocity component in MAC grid. If it's x velocity: $(i,j) = \left(\left\lfloor \frac{y-\frac{l}{2}}{l} \right\rfloor, \left\lfloor \frac{x}{l} \right\rfloor\right)$. If it's y velocity: $(i,j) = \left(\left\lfloor \frac{y}{l} \right\rfloor, \left\lfloor \frac{x-\frac{l}{2}}{l} \right\rfloor\right)$. And compute dx, dy with (i,j) and particle position.

```
    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];

            float gx, gy;
            if (component == 0) { // Vx component (located at (j*cell_dim, (i+0.5)*cell_dim))
                gx = pos.x() / cell_dim;
                gy = (pos.y() - 0.5f * cell_dim) / cell_dim;
            }
            else { // Vy component (located at ((j+0.5)*cell_dim, i*cell_dim))
```

```
            gx = (pos.x() - 0.5f * cell_dim) / cell_dim;
            gy = pos.y() / cell_dim;
        }
        int j_lower = static_cast<int>(gx);
        int i_lower = static_cast<int>(gy);

        float dx = gx - j_lower;
        float dy = gy - i_lower;
```

Compute the bilinear interpolation weights for the four neighboring grid points:

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

$$w_2 = \left(\frac{\Delta x}{l}\right)\left(1 - \frac{\Delta y}{l}\right) \tag{2}$$

$$w_3 = \left(\frac{\Delta x}{l}\right)\left(\frac{\Delta y}{l}\right) \tag{3}$$

$$w_4 = \left(1 - \frac{\Delta x}{l}\right)\left(\frac{\Delta y}{l}\right) \tag{4}$$

And store the weights in the array `weights` and store four sample points in array `v_indices`

```
        float w1 = (1.0f - dx) * (1.0f - dy);
        float w2 = dx * (1.0f - dy);
        float w3 = dx * dy;
        float w4 = (1.0f - dx) * dy;

        float weights[4] = { w1, w2, w3, w4 };

        Eigen::Vector2i v_indices[4] = {
            {i_lower, j_lower},
            {i_lower, j_lower + 1},
            {i_lower + 1, j_lower + 1},
            {i_lower + 1, j_lower}
        };
```

If we are transferring velocity from particles to cells: Get the velocity component from the particle and calculate which cell the particle is in. Mark the cell containing the particle as FLUID if it's not SOLID.

```
        if (to_cell)//粒子到網格
        {
            float v_p = (component == 0) ? particle_vel[i].x() : particle_vel[i].y();
            int p_cell_row = static_cast<int>(pos.y() / cell_dim);
            int p_cell_col = static_cast<int>(pos.x() / cell_dim);
            p_cell_row = std::max(0, std::min(p_cell_row, cell_rows - 1));
            p_cell_col = std::max(0, std::min(p_cell_col, cell_cols - 1));

            if (cell_types[p_cell_row][p_cell_col] != CellType::SOLID) {
                cell_types[p_cell_row][p_cell_col]=CellType::FLUID;
            }
```

For each of the four grid points, check if the grid point is within the valid bounds. If it's valid, add the weighed particle velocity to the grid point and accumulates the weights.

```
            for (int k = 0; k < 4; ++k) {
                int r_idx = v_indices[k].x();
                int c_idx = v_indices[k].y();

                bool valid_access = false;
                if (component == 0) {
                    valid_access = (r_idx >= 0 && r_idx < cell_rows && c_idx >= 0 && c_idx <=
                    ↪  cell_cols);
                }
                else {
```

```
                    valid_access = (r_idx >= 0 && r_idx <= cell_rows && c_idx >= 0 && c_idx <
                    ↪  cell_cols);
                }

                if (valid_access) {
                    cell_velocities[r_idx][c_idx][component] += weights[k] *v_p;
                    if (component == 0) weight_sum_x[r_idx][c_idx] += weights[k];
                    else weight_sum_y[r_idx][c_idx] += weights[k];
                }
            }
        }
```

If we are transferring velocity from cells to particles: Initialize the variable for PIC and FLIP.For the four sample points, check whether it's in the valid bounds.

```
        else
        {
            // TODO: Transfer valid cell velocities back to the particles using a mixture of PIC and
            ↪  FLIP
            //網格到粒子
            float v_p = (component == 0) ? particle_vel[i].x() : particle_vel[i].y();
            float pic_v_sum = 0.0f;
            float flip_delta_v_sum = 0.0f;
            float total_valid_w = 0.0f;

            for (int k = 0; k < 4; k++) {
                int r_idx = v_indices[k].x();
                int c_idx = v_indices[k].y();

                bool valid_access = false;
                if (component == 0) { valid_access = (r_idx >= 0 && r_idx < cell_rows && c_idx >= 0 &&
                ↪  c_idx <= cell_cols); }
                else { valid_access = (r_idx >= 0 && r_idx <= cell_rows && c_idx >= 0 && c_idx <
                ↪  cell_cols); }

                if (!valid_access) continue;
```

Check whether a velocity sample is valid. For MAC grids, a velocity component is valid if at least one of the adjacent cells is a FLUID cell.

```
                bool is_v_sample_valid = false;
                if (component == 0) { // Vx at (r_idx, c_idx) separates cell (r_idx, c_idx-1) and
                ↪  (r_idx, c_idx)
                    // Valid if c_idx is not a solid boundary itself and at least one adjacent cell is
                    ↪  FLUID
                    if (c_idx > 0 && c_idx < cell_cols) { // Interior Vx
                        is_v_sample_valid = (cell_types[r_idx][c_idx - 1] == CellType::FLUID ||
                        ↪  cell_types[r_idx][c_idx] == CellType::FLUID);
                    }
                    else if (c_idx == 0 && cell_types[r_idx][0] != CellType::SOLID) { // Left boundary
                    ↪  Vx (if not solid itself)
                        is_v_sample_valid = (cell_types[r_idx][0] == CellType::FLUID);
                    }
                    else if (c_idx == cell_cols && cell_types[r_idx][cell_cols - 1] !=
                    ↪  CellType::SOLID) { // Right boundary Vx
                        is_v_sample_valid = (cell_types[r_idx][cell_cols - 1] == CellType::FLUID);
                    }
                }
                else { // Vy at (r_idx, c_idx) separates cell (r_idx-1, c_idx) and (r_idx, c_idx)
                    if (r_idx > 0 && r_idx < cell_rows) { // Interior Vy
                        is_v_sample_valid = (cell_types[r_idx - 1][c_idx] == CellType::FLUID ||
                        ↪  cell_types[r_idx][c_idx] == CellType::FLUID);
                    }
                    else if (r_idx == 0 && cell_types[0][c_idx] != CellType::SOLID) { // Bottom
                    ↪  boundary Vy
                        is_v_sample_valid = (cell_types[0][c_idx] == CellType::FLUID);
                    }
                    else if (r_idx == cell_rows && cell_types[cell_rows - 1][c_idx] !=
                    ↪  CellType::SOLID) { // Top boundary Vy
```

```
                        is_v_sample_valid = (cell_types[cell_rows - 1][c_idx] == CellType::FLUID);
                    }
                }
```

Add the weighted velocity($w \cdot v$) to PIC calculation, and accumulate the weighted velocity change ($w(v - v_{prev})$) for FLIP calculation. Track the total weight of valid samples.

```
                if (is_v_sample_valid) {
                    pic_v_sum += weights[k] * cell_velocities[r_idx][c_idx][component];
                    flip_delta_v_sum += weights[k] * (cell_velocities[r_idx][c_idx][component] -
                    ↪  prev_cell_velocities[r_idx][c_idx][component]);
                    total_valid_w += weights[k];
                }
```

Calculate v_{pic} and v_{flip} by dividing the accumulated velocity to the accumulated sample weights. And compute final velocity by $v_p = \alpha_{flip} v_{flip} + (1 - \alpha_{flip}) v_{pic}$ where α_{flip} is flip_ratio parameter.

```
            if (total_valid_w > 0.0f) {
                float pic_v = pic_v_sum / total_valid_w;
                float flip_v_change = flip_delta_v_sum / total_valid_w;
                float flip_v = v_p + flip_v_change;

                float final_v = (1.0f - flip_ratio) * pic_v + flip_ratio * flip_v;
                if (component == 0) particle_vel[i].x() = final_v;
                else particle_vel[i].y() = final_v;
            }
```

Normalize the cell velocities by dividing it to the weight sum, and back up the velocity in `prev_velocity`.

```
    if (to_cell)
    {
        for (int i = 0; i < cell_rows; i++) for (int j = 0; j < cell_cols; j++) {
            float w = (component == 0) ? weight_sum_x[i][j] : weight_sum_y[i][j];
            if (w > 0.0f) {
                cell_velocities[i][j][component] /= w;
            }


            prev_cell_velocities[i][j][component] = cell_velocities[i][j][component];
        }
    }
```

2.3 Update Density

Reset the cell density.

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

Iterate over all particles. Find the base cell of the particle by $(i, j) = (\frac{\lfloor x - \frac{l}{2} \rfloor}{l}, \frac{\lfloor y - \frac{l}{2} \rfloor}{l})$, and compute `dx` and `dy`. Calculate weight of each sample point like how we compute when transfer velocity by using `dx` and `dy`.

```
    for (int i = 0; i < num_particles; ++i)
    {
        // TODO: Perform bilinear interpolation

        //the base cell of the particle
        float gx = (particle_pos[i].x() - 0.5f * cell_dim) / cell_dim;
        float gy = (particle_pos[i].y() - 0.5f * cell_dim) / cell_dim;

        int cellX = static_cast<int>(gx); // Column index of bottom-left cell center
        int cellY = static_cast<int>(gy); // Row index of bottom-left cell center

```

```
        float dx = gx - cellX;
        float dy = gy - cellY;

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

Check whether the cell index is valid, and add the density to the surrounding cells.

```
        if (cellY >= 0 && cellY < cell_rows && cellX >= 0 && cellX < cell_cols)
            cell_densities[cellY][cellX] += w1;

        if (cellY >= 0 && cellY < cell_rows && cellX + 1 >= 0 && cellX + 1 < cell_cols)
            cell_densities[cellY][cellX + 1] += w2;

        if (cellY + 1 >= 0 && cellY + 1 < cell_rows && cellX + 1 >= 0 && cellX + 1 < cell_cols)
            cell_densities[cellY + 1][cellX + 1] += w3;

        if (cellY + 1 >= 0 && cellY + 1 < cell_rows && cellX >= 0 && cellX < cell_cols)
            cell_densities[cellY + 1][cellX] += w4;
```

If the rest density hasn't been calculated, calculate the rest density: Sums up the densities of all cells marked as FLUID, and count the FLUID cells number. And update the rest density as `total_density / numFluidCells`. If there's no fluid cells, it defaults to 1.0.

```
    if (particle_rest_density == 0.0)
    {
        // TODO: Calculate resting particle densities in fluid cells.
        float total_density = 0.0f;
        int numFluidCells = 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];
                numFluidCells += 1;
            }
        }

        if (numFluidCells) {
            particle_rest_density = total_density / numFluidCells;
        }
        else {
            particle_rest_density = 1.0f;
        }
    }
```

2.4 Solving Incompressibility

Iterate each cell, and only apply the incompressibility constraint to FLUID cells, skip SOLID and AIR cells. Compute the divergence of fluid cell by $div = v_{top} + vright - v_{left} - v_{bottom}$ and calculate valid flow direction number. Skip if valid flow direction number is zero.

```
    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)
        {
            // TODO: Calculate divergence of fluid cells
            if (cell_types[i][j] != CellType::FLUID) continue;

            int flow_dirs = 0;
            float div = 0.0f;
            if (cell_types[i - 1][j] != CellType::SOLID) {//bottom
                div -= cell_velocities[i][j].y();
                flow_dirs++;
            }
```

```
            if (cell_types[i + 1][j] != CellType::SOLID) {//top
                div += cell_velocities[i + 1][j].y();
                flow_dirs++;
            }
            if (cell_types[i][j - 1] != CellType::SOLID) {//left
                flow_dirs++;
                div -= cell_velocities[i][j].x();
            }
            if (cell_types[i][j + 1] != CellType::SOLID) {//right
                div += cell_velocities[i][j + 1].x();
                flow_dirs++;
            }
            if (flow_dirs == 0) continue;
```

If density correction is enable and cells' density exceeds particle_rest_density, we need to correct divergence by $div = div - k_{stiff}(\rho_{cell} - \rho_{rest})$, where k_{stiff} is stiffness coefficient. Compute velocity correction by $v_{corr} = o \cdot \frac{div}{dirs}$ and o is over relaxation constant that can speed up convergence.

```
            if (density_correction && (cell_densities[i][j] > particle_rest_density)) {
                div -= stiffness_coefficient * (cell_densities[i][j] - particle_rest_density);
            }
            float corr = div / flow_dirs * over_relaxation;
```

Check if the neighboring cells are SOLID first. If not, add correction to the left and bottom velocities and subtract correction from top and right velocities.

```
            if (cell_types[i][j-1] != CellType::SOLID)
                cell_velocities[i][j].x() += corr;    // Left velocity

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

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

            if (cell_types[i+1][j] != CellType::SOLID)
                cell_velocities[i+1][j].y() -= corr;  // Top velocity
```

3 Results and Discussions

3.1 Cell Dimension

3.1.1 Observation

- Fluid moves faster and more realistically with small cell dimensions(5-15). Besides, it shows detailed turbulence and natural splashing.
- With medium cell dimensions(15-30), I observed the fluid motion become slower and it performed less detailed flow patterns.
- With large cell dimensions(30-50), the fluid flow becomes much slower, even making it look like some slow motion affects.

3.1.2 Explanation

- Small cells provide higher-resolution velocity field, preserving flow details
- Simulation with cell dimension within 5-10 can have better affect.

3.2 Flip Ratio

3.2.1 Observations

- **Low FLIP ratio (closer to 0):**

  - Water appears more smooth and cohesive
  - Reduced splashing and particle separation
  - During impact, fluid tends to move as larger cohesive masses
- **High FLIP ratio (closer to 1):**
  - More splashing effects
  - Water breaks into smaller, discrete particles during impacts
  - Creates more dynamic, spray-like behavior during rapid motion

3.2.2 Explanation

- FLIP ratio controls the blending between PIC and FLIP methods
- PIC (low ratio) averages velocities across the grid, creating smoother, more damped motion
- FLIP (high ratio) preserves particle velocity differences, maintaining more energy and detail
- Higher FLIP ratios preserve local vorticity and velocity gradients that cause separation
- Lower FLIP ratios introduce numerical diffusion that makes fluid simulation looks more smooth

3.3 Stiffness Coefficient

3.3.1 Observations

- **Low stiffness coefficient:**
  - Water particles appear more tightly packed
  - Creates a denser fluid appearance
- **High stiffness coefficient:**
  - Noticeable gaps form between water particles
  - Fluid expands and appears less dense

3.3.2 Explanation

- Stiffness coefficient controls the strength of density-based pressure forces
- Lower values allow fluid to compress more easily, maintaining higher densities
- Higher values aggressively enforce the incompressibility constraint, pushing particles apart
- Acts as a scaling factor for how strongly the simulation responds to density variations

4 Bonus

I enhance the visual realism of the fluid simulation by evaluating the particle color with it's height and velocity. Besides, I add the foam factor to simulate the foam affect of real fluid.

First, I calculate the velocity magnitude same as in the original. Then calculate a normalized height value (0 at bottom, 1 at top) which is new. And creates a height-based gradient effect for the water color:

- Red remains at 0(no red in water)
- Green ranges from 0.3 (at bottom) to 1.0 (at top)
- Blue ranges from 0.8 (at bottom) to 1.0 (at top)

This simulates how real water appears deeper blue at depth and lighter near the surface.

The `foamFactor` calculates how much "foam" to add based on velocity.Faster moving particles (high velocity) will have bigger foamFactor. `Eigen::Vector3f::Ones()` creates white color [1,1,1]. And blend the color between the base water color and white based on velocity.

```
    for (int i = 0; i < num_particles; ++i) {
        float vel = particle_vel[i].norm();
        float height = particle_pos[i].y() / viewport_h; // Normalized height

        // Blend between deep blue and light blue based on height
        Eigen::Vector3f waterColor = Eigen::Vector3f(0.0f, 0.3f + 0.7f * height, 0.8f + 0.2f * height);

        // Add white foam based on velocity
        float foamFactor = std::min(vel / 50.0f, 1.0f);
        particle_colors[i] = waterColor * (1.0f - foamFactor) + Eigen::Vector3f::Ones() * foamFactor;
    }
```

5 Conclusion

This assignment implemented a 2D fluid simulation using the PIC/FLIP method. The core components—particle relaxation, velocity transfer between particles and grid, density tracking, and incompressibility solving—work together to create realistic fluid behaviors.

Experiments with different parameters revealed that higher FLIP ratios (0.8-0.9) produced more dynamic, splashy water while lower values created more smooth flows. The cell dimension significantly impacted both performance and simulation, with smaller cells providing better resolution more realistic water vision. While simulation with big cell dimension creates weird water flow and makes it look like slow motion.

The additional implementation included a csolor enhancement that varies fluid particle colors based on height and velocity, creating a gradient from deep blue to light blue and adding white foam effects in high-velocity areas. This feature was implemented in the `updateParticleColors()` function by calculating normalized height and blending colors proportionally to velocity magnitude.