Project README
Title: 2D and 3D Fluid Simulator
Name: Honglin Cao
Student ID: 20882680
User ID: h45cao

Technical Outline:

Develop a flexible and efficient Position-Based Dynamics (PBD) framework

Originally, I aimed to develop a flexible and efficient Position-Based Dynamics (PBD) framework for fluid simulation. However, I found this approach to be overly complex for my current level of expertise. Instead, I opted for a force-based simulation framework, which offers a more precise simulation of fluid particles at the cost of requiring smaller time steps to maintain stability. The force-based framework is particularly well-suited for capturing the detailed interactions and behaviors of fluid particles, providing a high level of accuracy in the simulation.

The force-based framework can be described by the following steps, allowing for the development of both 2D and 3D fluid simulations:

- **ResetAcceleration()**: Initializes the accelerations of particles, setting them to the influence of gravity. This step ensures that each particle starts with a base acceleration due to gravitational forces.
- **GridSearch()**: Organizes particles into a grid-based spatial partitioning system. This step facilitates efficient neighbor searching by dividing the simulation space into smaller blocks, reducing the computational complexity of finding neighboring particles.
- **DensityAndPressure()**: Computes the density of each particle based on the influence of its neighbors. The pressure of each particle is then updated using the calculated densities. This step is crucial for simulating realistic fluid dynamics as it models how particles compress and expand.
- **ViscosityAcceleration()**: Calculates the changes in particle accelerations due to viscosity forces. Viscosity forces account for the internal friction between fluid particles, affecting how they flow and interact with each other.
- **PressureAcceleration()**: Updates particle accelerations based on pressure forces. These forces are derived from the differences in pressure between particles, driving fluid motion and ensuring realistic fluid behavior.
- **MoveStep()**: Integrates the velocities and positions of particles using the calculated accelerations. This step updates the particle states, moving them according to the applied forces. It includes time step constraints to maintain numerical stability.
- **CheckBoundary()**: Applies boundary conditions to particles, ensuring they remain within the simulation domain. Particles that reach the boundaries are reflected, preventing them from leaving the simulation area and maintaining the integrity of the simulation space.

Implement Smoothed Particle Hydrodynamics (SPH) for fluid simulation

[1]

I implemented the most simple version of Smooth Particle Hydrodynamics, called Weakly Compressible Smooth Particle Hydrodynamics (WCSPH). Intuitively, we can think it as using penality force to handle the overlap of water particles.

I implemented the Smoothed Particle Hydrodynamics (SPH) based on my force-based framework. This involved developing SPH algorithms to simulate realistic fluid behavior, implementing kernel functions, and solving the Navier-Stokes equations to model fluid dynamics accurately.

In my implementation, I use a Cubic Spline Kernel. The Cubic Spline Kernel is a popular choice for SPH simulations due to its smoothness and compact support. The kernel function W is defined such that it has a finite range, making computations efficient. The implementation is in class `CubicSplineKernel2D` and `CubicSplineKerne3D`.

[1] I will only cover the detail of implementation in 2D, as the 3D is very similar.

- **Cubic Spline Kernel**: The cubic spline kernel function in 2D, $W(\mathbf{r}, h)$, is defined as:

$$W(\mathbf{r}, h) = \frac{40}{7\pi h^2} \times \begin{cases} 1 - 6\left(\frac{|\mathbf{r}|}{h}\right)^2 + 6\left(\frac{|\mathbf{r}|}{h}\right)^3 & \text{if } 0 \leq \frac{|\mathbf{r}|}{h} < 0.5 \\ 2\left(1 - \frac{|\mathbf{r}|}{h}\right)^3 & \text{if } 0.5 \leq \frac{|\mathbf{r}|}{h} < 1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

  where:

  - $|\mathbf{r}|$ is the distance between particles
  - h is the smoothing radius

  The gradient of the cubic spline kernel in 2D, $\nabla W(\mathbf{r}, h)$, is given by:

$$\nabla W(\mathbf{r}, h) = \frac{40}{7\pi h^2} \times \begin{cases} \left(-12\frac{|\mathbf{r}|}{h^2} + 18\frac{|\mathbf{r}|^2}{h^3}\right)\frac{\mathbf{r}}{|\mathbf{r}|} & \text{if } 0 \leq \frac{|\mathbf{r}|}{h} < 0.5 \\ -6\left(1 - \frac{|\mathbf{r}|}{h}\right)^2 \frac{\mathbf{r}}{|\mathbf{r}|} & \text{if } 0.5 \leq \frac{|\mathbf{r}|}{h} < 1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

- **Density**: The density ρ_i of particle i is computed by summing the contributions from neighboring particles within a support radius using the SPH interpolation kernel:

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3}$$

  where:

  - m_j is the mass of particle j
  - W is the SPH kernel function
  - $\mathbf{r}_i - \mathbf{r}_j$ is the distance between particles i and j
  - h is the smoothing radius

- **Pressure**: The pressure P_i of particle i is derived using the equation of state:

$$P_i = k(\rho_i - \rho_0) \tag{4}$$

  where:

  - k is the stiffness constant
  - ρ_0 is the reference density

- **Navier-Stokes Equations for Particle-based Fluids**

  The Navier-Stokes equations describe the motion of fluid substances and are solved for particle-based fluids as follows:

  - **Momentum Equation**

    The momentum equation for particle i is given by:

$$\frac{d\mathbf{v}_i}{dt} = -\frac{1}{\rho_i}\nabla P_i + \nu\nabla^2\mathbf{v}_i + \mathbf{g} \tag{5}$$

    where:

* $\mathbf{v}_i$ is the velocity of particle i
* ∇P_i is the pressure gradient
* ν is the kinematic viscosity
* $\nabla^2 \mathbf{v}_i$ is the Laplacian of velocity
* $\mathbf{g}$ is the gravitational acceleration

– **Discretization using SPH**

In SPH, the pressure gradient and viscosity terms are discretized as follows:

$$\nabla P_i = \rho_i \sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6}$$

$$\nabla^2 \mathbf{v}_i = 2d\nu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \frac{W(\mathbf{r}_i - \mathbf{r}_j, h)}{\mathbf{r}_i - \mathbf{r}_j} \tag{7}$$

where d is the number of spatial dimensions.

- **Boundary**

Reflective boundary conditions are applied to keep particles within the simulation domain, ensuring particles bounce back when they hit the boundaries:

$$\mathbf{v}_i = -\mathbf{v}_i \quad \text{if particle hits boundary} \tag{8}$$

Modify the rendering engine to use OpenGL.

1. **Initialize OpenGL and Create a Window:**
   - Initialize the GLFW library to manage the window and OpenGL context.
   - Set the appropriate GLFW window hints for the desired OpenGL version and profile.
   - Create a window with a specified width, height, and title.
   - Make the window's context current and load all OpenGL function pointers using GLAD.
   - Set up GLFW callback functions for handling errors, key inputs, mouse movements, and window resizing.
   - Enable depth testing for correct rendering of 3D objects.
2. **Define Camera Parameters and Movement:**
   - Define camera parameters such as position, direction, up vector, yaw, pitch, field of view (FOV), and movement speeds.
   - Implement callback functions to handle key inputs for camera movement and mouse movements for adjusting camera orientation.
   - Update the camera's front vector based on the yaw and pitch angles calculated from mouse movements.
3. **Generate Sphere Geometry:**
   - Define a vertex structure to hold position and color data.
   - Implement functions to generate vertices and indices for a sphere based on specified radius, sector count, and stack count.

- Create buffers for the sphere, including vertex array object (VAO), vertex buffer object (VBO), and element buffer object (EBO).
- Bind the buffers and set up vertex attribute pointers for position and color.

4. **Compile and Link Shaders:**
   - Implement functions to compile vertex and fragment shaders from source files.
   - Create a shader program by attaching and linking the compiled shaders.
   - Check for compilation and linking errors, and output appropriate error messages.
5. **Render the Scene:**
   - Implement the main render loop, which continues until the window should close.
   - Clear the color and depth buffers at the beginning of each frame.
   - Set up the view and projection matrices based on the camera parameters.
   - For each particle in the particle system, calculate the model matrix and pass it to the shader program.
   - Bind the vertex array object and draw the sphere using the index buffer.
   - Swap the front and back buffers to display the rendered frame, and poll for and process events.
6. **Clean Up Resources:**
   - Delete the shader program and vertex array object.
   - Destroy the GLFW window and terminate the GLFW library to clean up resources.

Optimizing Data Structures for Efficient Simulation

- **Grid-based Spatial Partitioning**

  The simulation domain is divided into a grid, where each cell contains a list of particles. The size of each cell is determined by the support radius h of the SPH kernel. Particles are assigned to cells based on their positions.

  1. **Initialize Grid:** Create a grid with cell size equal to the support radius h.
  2. **Assign Particles to Cells:** For each particle, compute the cell index based on its position and add the particle to the corresponding cell.
  3. **Neighbor Search:** For each particle, identify neighboring particles by checking the current cell and the adjacent cells.

  The grid index for a particle at position $\mathbf{r}$ is computed as:

$$\mathbf{c} = \left\lfloor \frac{\mathbf{r}}{h} \right\rfloor \tag{9}$$

  where $\mathbf{c}$ is the cell index.

- **Neighbor Information Data Structure**

  The neighbor information for each particle is stored in a structure that includes the index of the neighboring particle, the distance between particles, the squared distance, and the radius vector:

```
struct NeighborInfo2D {
    int particleIndex;
    float distance;
    float distanceSquared;
    glm::vec2 radiusVector;
};
```

Ensure stability and performance optimization

- **Time Integration**

  The velocity and position of particles are updated using a time-stepping method such as the Euler or Runge-Kutta method. The time step Δt is chosen to satisfy the Courant-Friedrichs-Lewy (CFL) condition for stability:

$$\Delta t \leq \frac{h}{\max_i(|\mathbf{v}_i|)} \tag{10}$$

- **Artificial Viscosity**

  Artificial viscosity is introduced to handle shock waves and prevent particle interpenetration:

$$\Pi_{ij} = \begin{cases} \frac{-\alpha \bar{c} \mu_{ij} + \beta \mu_{ij}^2}{\bar{\rho}_{ij}} & \text{if } \mathbf{v}_{ij} \cdot \mathbf{r}_{ij} < 0 \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

  where:

  - α and β are parameters controlling the artificial viscosity
  - $\bar{c}$ is the average speed of sound
  - $\mu_{ij} = \frac{h(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})}{|\mathbf{r}_{ij}|^2 + \epsilon h^2}$
  - $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$
  - $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$
  - $\bar{\rho}_{ij}$ is the average density
  - ϵ is a small number to prevent division by zero

- **Performance Optimization**

  I employed the strategy I used in A3, where I divided the area of particles into grids, and computed the neighbour particle (particles inside the same grid) info for each particles inside each grid; And dispatch as many threads as possible to at each stage of function `Iterate()`. To achieve this, the information of all particle is stored as vector inside `ParticleSystem` to support addition and remove of particles at runtime.