

2D Interactive Fluid Simulator Final Report

Marissa Liu

d

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
mhliu@mit.edu

Otto Beall

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
o_beall@mit.edu

Abstract—This project is a 2D particle-in-cell (PIC) fluid simulator with real time response to user interaction. Fluid simulations are computationally intensive, but implementation in hardware enables parallelization that may facilitate higher performance. Implemented using an AMD Artix-7 FPGA, the simulation will display the movements of 4000 particles on a 720x720 TFT display with a target update rate of 60Hz. A frame buffer in DRAM will be used to store the simulation output and an accelerometer will be used to measure the tilt of the board.

I. POSITION UPDATE MODULES (MARISSA)

Updating particle positions involves not only the advection of the particles based on velocity and acceleration, but also the organization of particles based on approximate position and the management of collisions with the border. Thus, the design has been broken up into three distinct modules: particle advection, collision management, and spatial hashing.

A. Particle Advection

To update particle positions, it is necessary to iterate through each particle, read its position and velocity BRAMs, and calculate a new position based on these values. The x and y positions can be treated independently, and the updated positions are simply be the sum of the previous position and the velocity along the same axis.

At the same time as the write to the position BRAMs, acceleration measurements from the accelerometer module are be used to update particle velocities. The updated velocities will be a sum of the previous velocity and acceleration along the same axis.

B. Spatial Hashing

Since the particle position and velocity BRAMs are addressed based on particle id (a number ranging from 0-3999) to access to a specific particle's position and velocity in both x and y, it is not possible to access particles based solely on approximate location. The solution is spatial hashing, which requires dividing the space into a 90x90 grid and sorting particles based on which grid cell they lie in. Updated every refresh cycle, this approximates a sorted list of particles based on x and y location. Spatial hashing is necessary for the display module and collision management to work in fewer clock cycles.

This module introduces four new BRAM structures as outputs of spatial hashing. The first is sorted particle id

BRAM, which stores the particle ids (0-3999) in order based on their grid cell locations. The next two are sorted x and y position BRAMs that correspond to the positions of particles in the sorted particle id BRAM. The final BRAM stores cell pointers, indicating at which address in the previous BRAMs holds the first particle in the grid cell. The cell pointer also hold 2 bits that indicate how many particles are in the cell.

There are three generalized steps to spatial hashing, which will correspond to three states in the module's FSM:

- 1) **Count Particles:** Iterate through all particles and their positions, determine which grid cell they belong to (divide by 8 or right shift by 3), and count how many particles per grid cell. These counts are stored in a BRAM used only by the module.
- 2) **Calculate Partial Sums:** Iterate through all grid cells and add total particles seen in current and previous cells, store partial sum in cell pointer BRAM.
- 3) **Update Pointers and Sort:** Iterate through all particles and their positions again. Again, determine which grid cell they belong to. Write position and cell ids into the sorted BRAMs based on cell pointer BRAM value for the given cell. Decrement the cell pointer value by one and write back to the same address in BRAM.

Additional states include an idle state while the module waits for a start command and a zeroing state, where the module zeros out its particle count and the cell pointer BRAMs. One challenge to implementing this module was dealing with the instance in which particle shared a grid cell with a particle already in the pipeline, though this was dealt with by using a bypass that checked for this case.

The spatial hashing module requires 24,200 clock cycles to complete, which was confirmed with our Cocotb Testbench. The additional BRAMs require 417.4 kbits, but without hashing, to draw particles or determine collisions you would be required to read all 4000 particle positions over and over again, quickly adding up. This becomes a trade off of more memory usage for fewer clock cycles.

Hashing was successfully verified through testbenching.

II. DISPLAY MODULES (MARISSA)

The display portion is split into three parts - the logic for drawing particles, interacting with the DRAM frame buffer, and outputting to the TFT display.

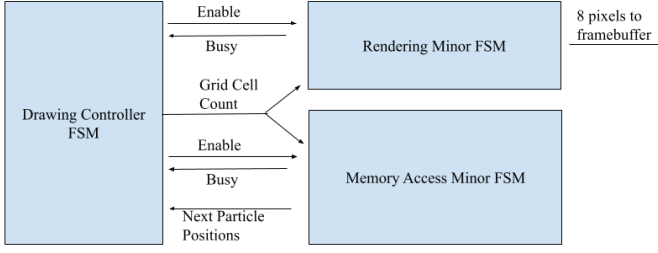


Fig. 1. Diagram of minor and major FSMs involved in the drawing module

A. Drawing Particles

Particles with a radius of four are drawn, one grid cell at a time by checking the active grid cell and its 8 surrounding grid cells for particles and their positions, which are effectively the center points of the particle. Checking surrounding grid cells is necessary as particle areas may leak into adjacent cells. When complete, the new active grid cell rolls over to be the one to the right of the current one. This requires reading the next three grid cells' particles and their positions, but allows for the reuse of six cells.

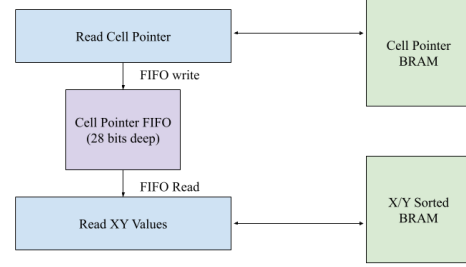
Thus, the Drawing module uses two minor FSMs - one to handle the logic of producing the shape within the active grid cell, and another to load in the positions of the particles in the next cells that will be checked. These run in parallel to minimize clock cycle usage and are controlled by a major FSM that increments to the next active cell when both minor FSMs are finished.

The Memory Access Minor FSM determines what the next cells that need to be checked based on the active grid cell are, and then reads from both the particle cell pointer BRAM and sorted X/Y position BRAMs to determine where the particles in those cells are. This requires attention to edge cases (active grid cells on the rightmost side of the screen as well as the top and bottom) as well as an additional buffer to account for the fact that a grid cell can have a variable amount of particles inside it. Thus, a FIFO buffer is written to with the grid cell pointer information along with metadata, and then it is read from when the parallel process reading position values is ready. The included metadata allows the process on the read side of the FIFO to know which grid cell register to write the new x and y positions in. This enables the pipeline on both ends to run more smoothly and without stalls.

The Particle Rendering Minor FSM takes the position values, finds the distance between the particles and the center of the active grid cell, shifts a 64 bit template in the shape of a pixel based on these values, then colors the pixels in. From there, the completed 8x8 pixel square is sent to the frame buffer one row (8 pixels) at a time.

B. Frame Buffer

The output of the drawing module is sent directly to the frame buffer modules. Though the drawing module can render 64 pixels at a time, it outputs in 8 8-pixel chunks serially.



FIFO data structure: {cell pointer[27:4], which cell [3:1], last cell [0]}

Fig. 2. Drawing Memory Access Module Structure

These 8 pixel chunks are accompanied by their DRAM address, and is sent to a modified high definition frame buffer module in the following 128 bit data structure:

address[127:112], 14-bit rgb pixels [111:0]

From there, the address and pixel data are placed into an AXIS FIFO to interface between the 100MHz clock of the FPGA and the 83.33MHz clock of the DRAM controller and also handle the variation in clock cycles between outputs of the drawing module. The FIFO is read and its output is written to DRAM memory using the DRAM memory controller. Another AXIS FIFO is used on the output to store reads from the DRAM, and is synced with the TFT display. The output of this FIFO is then unstacked to be sent to the TFT controller pixel by pixel. This is essentially a modified version of lab 6, but because we write to DRAM non-sequentially we send to address through the AXIS FIFO as well.

C. TFT Display

The TFT inputs include a pixel clock, vsync, hsync and 18 parallel RGB data pins. Thus, the output to the TFT is be relatively straightforward as the output of the frame buffer can be split into bits and output at the same time. A video signal generator module drives the the display module's timing.

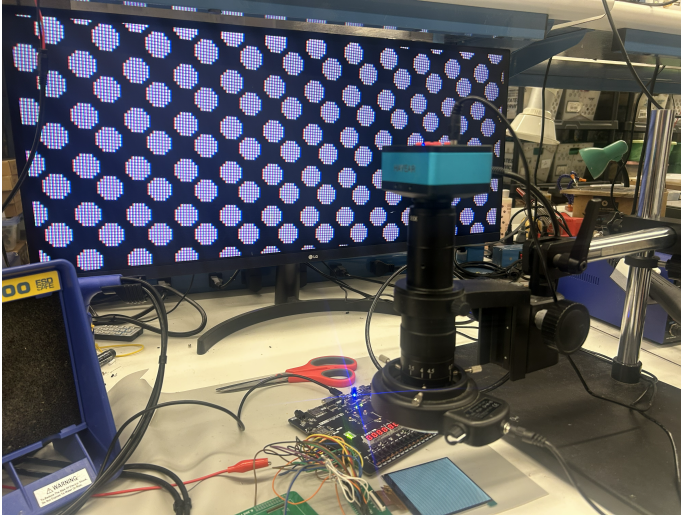


Fig. 3. Image of rendered particles on tft under the microscope - you can see each pixel

III. TESTBENCHING (OTTO)

Rather than implement the fluid simulator in SystemVerilog directly, the algorithm has been developed in Python to verify that the fluid simulator functions as expected. The testbench was first implemented with floating-point math, then retrofitted with 32-bit and 64-bit integers using the NumPy library. The visualization was produced using the PyGame library. The simulator slows down below 60Hz for particle counts greater than 100, so testing was performed with 100-400 particles. Faster performance with more particles will be achievable on the FPGA.

The testbench performs the following steps repeatedly inside the main loop:

- 1) **Draw Particles**
- 2) **Advect Particles**
- 3) **Handle Particle-Particle Collisions**
- 4) **Handle Particle-Boundary Collisions**
- 5) **Update Particle Density**
- 6) **Convergence Iterations to Solve Incompressibility**

These steps follow the sequence of modules which are activated sequentially on the FPGA. The final working integer-based simulation is called `fluid_simulation_int.py`.

A. Drawing Particles

In the Python testbench, the particles are drawn simply by iterating through each of the particles and calling the PyGame `draw.circle()` function which creates a circle of a specified radius at the particle's location. In the Python testbench, the particle X positions and Y positions are stored in separate one-dimensional arrays of 32-bit signed integers, analogous to the BRAM arrays which will store the particle X positions and Y positions on the FPGA. The particle drawing function translates from raw position value to pixel location by right-shifting by the parameter `POS_PIXEL_SHIFT` which has a value of 15. In effect, the particle's position value gets divided by 2^{15} to obtain the particles pixel position. In other words,

there are 2^{15} possible positions the particle can occupy within each pixel. In practice, a particle in any of these positions is drawn in the same way, but the increased precision of the particle's location allows for more fine-grained calculations for fluid divergence.

In SystemVerilog, the module for drawing the particles is more complex than in the testbench, as the module cannot simply read the entire frame buffer and modify to add a single circle one at a time. Instead, the particles are hashed based on the section of the screen they occupy, and only segments of the particle buffer need to be accessed and updated at a time.

B. Particle Advection

After the particles are drawn, the particle positions are updated according each particle's velocity in the x- and y-direction. Like the position values, both X and Y velocity are stored in separate one-dimensional arrays of signed 32-bit integers. The velocities are also updated according to the gravitational acceleration vector. On the FPGA, this vector is obtained from an accelerometer, but in the testbench, the direction and magnitude of the gravitational acceleration vector are obtained from the relative position of the mouse from the center of the drawing window. Thus, by moving the mouse, the testbench user can simulate tilting the screen and watch the fluid respond in real time.

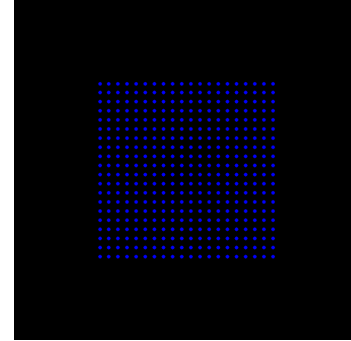


Fig. 4. Initial configuration of 100 fluid particles

C. Particle-Particle Collision Handling

This section of the fluid simulation algorithm was not a part of the original plan, but proved to be necessary for a realistic fluid simulation. In the real world, two fluid particles cannot occupy the same spatial position at the same time. Because individual fluid particles can be distinguished in our fluid simulation, the overlapping of particles creates an unrealistic effect. The incompressibility algorithm, which ensures that the fluid has a roughly constant density of particles, does not guarantee that particles will never overlap. The `pushParticleApart()` function modifies the particle positions to ensure that no two particle centers are closer than a particle diameter apart. A naive implementation of this function would check all pairs of particles for overlapping, but this implementation would scale quadratically with the number of particles. Instead, a spatial hashing function was used to sort the particles into a

rectangular grid of cells whose sidelength equals the diameter of each particle. Thus, each particle only needs to be checked against the particles in the 8 adjacent cells. If a collision is detected with another particle, the two particles are pushed apart along equal and opposite vectors to ensure they become just adjacent to one another. Multiple iterations of this process are needed to ensure convergence. The number of iterations is determined by the variable `PARTICLE_ITERS`, and a value of 5 was found sufficient to achieve convergence.

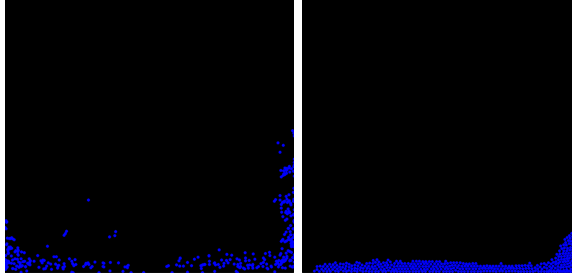


Fig. 5. Simulation without (left) and with (right) particle-particle collision handling. Note the overlapping particles in the left image

D. Particle-Boundary Collision Handling

In the process of particle advection and collision handling, some particles may be pushed outside the bounds of the simulation. This function simply checks whether any particles have exited the bounds of the simulation and moves them to be immediately within the bounds of the simulation. Because this function occurs after the particle-particle collision handling, it may result in some overlapping particles at the boundaries of the simulations, but since these overlaps only occur at the boundaries, the result does not detract from the overall verisimilitude of the simulation.

E. Update Particle Density

To maintain the incompressibility of the cells, the simulation space is divided into a grid of 45 by 45 square cells. (Note that this is a different grid from the spatial hashing grid used for particle-particle collision handling). The grid densities are stored as unsigned integers in a one-dimensional array which is $45 \times 45 = 2025$ elements long, and initialized to all zeros at the beginning of the calculation. This function iterates through each of the particles and finds the cell grid it is located in by dividing its pixel location by 16, as each grid cell is 16 pixels wide. Since 16 is 2^4 , this calculation can be performed as a right shift of 4 bits. (Note that this bit shift is performed on the pixel position value, which is itself right-shifted from the raw pixel position value). The density value of each grid cell is incremented by 1 for each particle contained in that cell. Density is measured in units of particles per grid cell.

F. Convergence Iterations to Solve for Incompressibility

This calculation is mostly unchanged from the original plan. First, the particle velocities are transferred as weighted averages to the centers of the nearest cell boundaries, which

store the net fluid velocity through the boundary. Next, the solver repeatedly calculates the divergence (net outflow of fluid) of each grid cell, and adjusts the velocities of each cell boundary to satisfy the condition of zero divergence for each grid cell. The number of iterations for this calculation is stored in the variable `CONVERGENCE_ITERS`. Finally, the velocities of the cell wall are transferred back to the particles. The X and Y velocities of each particle are calculated to be weighted averages of the four nearest cell boundaries. 10 iterations were found to be sufficient to achieve reasonable convergence.

IV. PARTICLE ADVECTION MODULE (OTTO)

The hardware implementation of the particle advection system is responsible for changing the particle's positions according to their velocities. The velocities and positions are both stored in separate BRAMs. The particle advection module has a signal which iterates from 1 to 4000 and reads into both the particle position and velocity BRAMs. The sums of the positions and velocities are then written into the particle BRAM. This module was completed but never tested.

V. BOUNDARY CHECKER (OTTO)

The boundary checker is responsible for checking which particles are outside the boundaries of the screen, and moving them back into place. Similar to the Particle Advection module, this module produces a signal which iterates from 1 to 4000, and checks whether the particle positions are greater than 720 or less than 0. The particles are then moved into the nearest position which is correctly in the boundaries of the screen. This module was not completed.

VI. PARTICLE HANDLER (OTTO)

The particle handler was intended to handle all the operations of the fluid simulator outside of rendering the particles. It contains both the Boundary Checker module and the Particle Advection module. This module was not completed.