Real-Time Fluid Simulation

Advanced Computer Graphics Spring 2020

Zachary Smeton

***Abstract*—Simulated fluids can produce incredible visuals and more believable interactions. The development of algorithms and techniques to perform these simulations in real-time interactive environments poses many challenges. But recent works on Smoothed Particle Hydrodynamics and Position Based Dynamics have been able to produce compelling results. I walk through the process of implementing the work of Miles Macklin and Matthias Müller on Position Based Fluids in OpenGL. My implementation generates realistic fluid simulations in a variety of situations but unfortunately was not able to achieve real-time speeds.**

***Index Terms*—position based dynamics, SPH, graphics, fluid simulation**

I. Introduction

Fluid simulation, in general, is a widely researched topic of computer graphics. A proper fluid simulation, at least for graphics, can capture the beauty and complexity that fluids have within the physical world. From ocean scapes to slime guns, fluids can create incredible scenes for video games and movies alike. The beauty of the final product is dependent on the quality of the simulation.

Fluid motion can be described nearly perfectly using the Navier-Stokes equations; however, these equations are incredibly complicated to solve and usually require the use of supercomputers to effectively numerically approximate [1]. As such, either major simplifications have to be made or different equations and techniques must be used to model fluids efficiently.

For this project, the goal was to implement a fluid simulation that works in non-bounded environments, supports collision with complex objects, and is suitable for real-time applications such as video games. This way the resulting implementation could be used in a wide variety of situations from waves crashing into a lighthouse or water being shot from a water gun to toxic sludge pouring out of a pipe.

In the following sections, I will discuss a few of the different techniques that focus on solving this type of simulation, then I will expand on the problem definition and what challenges must be overcome. I will dive into the details of my solution, and finally, I will discuss the results of my fluid simulation and future work.

II. Related Work

Existing techniques vary widely in which components of the fluid they are attempting to model, be it movement, interaction, or appearance. Many techniques focus on specific applications like height fields for large waterscapes [2] or particle simulations for smaller quantities of liquid with more interaction [3]. They usually focus on different types of fluids like water, smoke, fire, explosions, syrup. I will go overworks that focus on particle-based simulations focused mostly on simulating water.

One of the most well known and commonly used techniques is Smoothed Particle Hydrodynamics (SPH) which was proposed in [3]. SPH is a general approach to modeling physical systems. It represents the system as a set of particles, interpolates information about an individual particle from its neighbors, and then calculates forces to be applied to that particle. [3] goes over how SPH can be used to model gas dynamics, stellar collisions, planetary impact, cloud collisions, motion near black holes, and nearly incompressible flow (fluids). SPH can create very realistic fluid simulations, however, SPH is not well suited for interactive simulations. SPH will become unstable due to neighbor deficiencies and therefore require very small time steps or lots of particles to avoid this issue, resulting in high computational complexity.

To combat the issue of large time steps Miles Macklin and Matthias Müller developed a new approach that uses many of the concepts from SPH but uses the framework of Position Based Dynamics (PBD) to try to alleviate the issues caused by neighbor deficiencies in [4]. The result is Position Based Fluids (PBF). PBF uses the same density estimation techniques used by SPH but then updates the particle positions by solving a system of constraints such that there is uniform density. As position updates are less susceptible to instabilities than force-based approaches the resulting simulation allows for much larger time steps to be used [4]. This makes PBF much better for real-time graphics. Although their implementation is not perfect. There are issues with incorrect density estimations when interacting with objects causing particles to stack along object boundaries, the Jacobian solver used is slow to converge and could be sped up with a more sophisticated solver, and finally, their implementation has a lot of dependent parameters which makes parameter tuning difficult [4].

III. Problem Statement

The goal of my project is to implement a particle-based fluid simulation that can be used in real-time graphical applications. To do this I will implement PBF in OpenGL using compute shaders. Although the general algorithm has already been laid out for the project, there are many nitty-gritty details I will have to find solutions for and implement. The biggest challenge throughout all of this will be finding algorithms that are efficient (O(n), where n is the number of particles).

The first hurdle to overcome will be to implement neighbor finding, The naive approach is polynomial and is not fast enough for a real-time environment. Next will be implementing the particle simulation. This task will focus on how to split up the algorithm outlined in into different shaders and what graphical objects will be used to store the results. The final problem will be figuring out how to perform collision detection and response with external objects.

As this project is already a monumental task for a single individual, I will not attempt to solve any of the underlying issues in [4]'s implementation. Realistic fluid rendering will also not be a goal of this project and instead, the particles will just be rendered as spheres. Finally, the fluid will collide and respond to the object, but the object will not be affected by the fluid collision.

IV. Approach

Before diving into the specifics of the simulation loop and its implementation in OpenGL, having some background on the mathematics that control the whole simulation is vital. In order to enforce incompressibility PBF solves a system of constraints such that each particle has the same density, the rest density ρ_0. The system of constraints consists of a constraint function for each particle. Following [4], the constraint function for the ith particle is defined as:

$$C_i(\mathbf{p}_1, ..., \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1 \tag{1}$$

Where $\mathbf{p}_1$,...,$\mathbf{p}_n$ are the positions of the ith particle and the positions of its neighbors. The density of the ith particle, ρ_i, is calculated using the SPH density estimator [4]:

$$\rho_i = \sum_j m_j * W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{2}$$

Where h is the support radius and $W(\mathbf{p}_i - \mathbf{p}_j, h)$ is the Poly6 smoothing kernel which can be found in [5]. Each simulation loop we then calculate a change in particle positions $\Delta\mathbf{p}$ such that [4]:

$$C(\mathbf{p} + \Delta\mathbf{p}) = 0 \tag{3}$$

To do this we can take a series of steps along the constraint gradient [4]:

$$\Delta\mathbf{p} \approx \nabla C(\mathbf{p})\lambda \tag{4}$$

$$C(\mathbf{p} + \Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla C^T \Delta\mathbf{p} = 0 \tag{5}$$

$$\approx C(\mathbf{p}) + \nabla C^T \nabla C \lambda = 0 \tag{6}$$

Solving this for λ gives:

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2} \tag{7}$$

Where $\nabla_{\mathbf{p}_k} C_i$ is the gradient of the constraint function with respect to particle k, the equation for this can be found in [4].

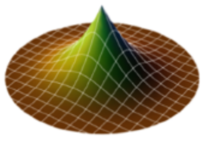


Fig. 1: Gradient of Spiky Smoothing Kernel [6]

Due to the fact that the constraint function (1) has a vanishing gradient at the smoothing kernel boundary, which can be seen in Figure 1. The denominator in (7) will approach 0 when a particle is separating from the rest of the fluid causing λ_i to approach ∞. To avoid this a user defined variable ε is introduced. Resulting in the following equation [4]:

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2 + \varepsilon} \tag{8}$$

The resulting position update is:

$$\Delta\mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{9}$$

Where ∇W is the gradient of the Spiky Smoothing Kernel as defined in [5].

Clumping and clustering of particles due to neighbor deficiencies is one of the issues that [4] addresses. [4] follows the approach of [7]. Which adds in an artificial pressure term defined as:

$$s_{corr} = -k(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta\mathbf{q}, h)})^n \tag{10}$$

Where $\Delta\mathbf{q}$ is a point at a user defined distance inside the smoothing kernel radius, and k is a small positive constant. Including this into the position update, (10) becomes:

$$\Delta\mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{corr}) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{11}$$

This the effect of particles repelling each-other when they get too close by introducing artificial pressure, reducing the amount of clustering, see Figure 2.

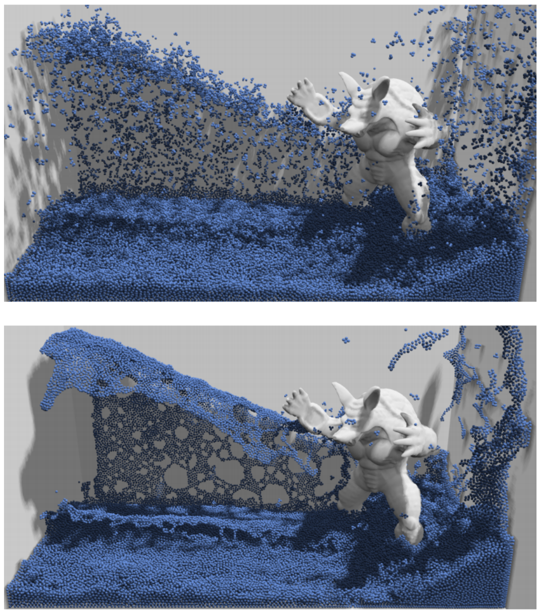


Fig. 2: "Armadillo Splash, Top: particle clumping due to neighbor deficiencies, Bottom: with artificial pressure term, note the improved particle distribution and surface tension." [4]

With the math for the position update covered, we can dive further into the fluid simulation. As stated previously, I will be following the fluid simulation proposed in [4]. See Algorithm 1 for the pseudo-code of the simulation loop. The algorithm can be split into four parts: position prediction, neighbor find, constraint solving, and velocity update. I will walk through each of the steps and lose data structures behind each of them and then I will dive into the details of how it was implemented using OpenGL.

Algorithm 1: Simulation Loop [4]

```
forall particles $i$ do
    apply forces $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_{ext}(\mathbf{x}_i)$
    predict position $\mathbf{x}_i^* \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
end
forall particles $i$ do
    find neighboring particles $N_i(\mathbf{x}_i^*)$
end
while iter < solverIterations do
    forall particles $i$ do
        calculate $\lambda_i$
    end
    forall particles $i$ do
        calculate $\Delta \mathbf{p}_i$
        perform collision detection and response
    end
    forall particles $i$ do
        update position $\mathbf{x}_i^* \leftarrow \mathbf{x}_i^* + \Delta \mathbf{p}_i$
    end
end
forall particles $i$ do
    update velocity $\mathbf{v}_i \leftarrow \frac{1}{\Delta t}(\mathbf{x}_i^* - \mathbf{x}_i)$
    apply vorticity confinement and XSPH viscosity
    update position $\mathbf{x}_i \leftarrow \mathbf{x}_i^*$
end
```

A. Position Prediction

Position prediction is relatively simple, update the velocity and position of each particle using forward Euler integration of external forces, which for this simulation is only gravity.

B. Neighbor Finding

Neighbor finding then takes the updated positions of all of the particles and then computes for each particle which particles are within its support radius (the distance at which a particle is considered a neighbor). Neighbor finding is one of the most computationally expensive parts of the simulation. So the algorithm used to perform neighbor finding must be highly parallelizable and be spatially and computationally efficient for this simulation to be interactive.

One common technique is to use a uniform grid. A uniform grid stores a list of particles that are in each of the grid cells, where the size of the grid cell is the support radius, this way each particle just has to check its cell and the 26 adjacent cells for its neighbors [5], see Figure 3. Using uniform grids for neighbor search has a computational complexity of O(n) because PBF moves particles to ensure there is a uniform density meaning each cell will contain the same number of particles. However, uniform grids are not the best choice because to use them the fluid must be restricted to within the bounds of the grid.

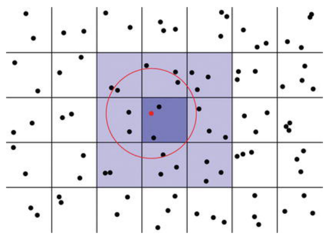


Fig. 3: Neighbor search in a uniform grid [8]

To enable the fluid to be unconstrained a spatial hash table is used instead. A spatial hash table uses a hashing function which maps each particle's position to a grid cell and then hashes the grid cells coordinates to a unique index [5]. The spatial hashing function can be seen in the equation (12). Where h is the support radius, m is the table size, and p_1, p_2, p_3 are all large prime numbers [5].

$$hash(x,y,z) = [(\frac{x}{h} * p_1) xor (\frac{y}{h} * p_1) xor (\frac{z}{h} * p_3)] \% m \quad (12)$$

Using the spatial hash we construct two lists: a hash table and a linked list. The hash table stores the index of the first particle in the linked list. The linked list entries store a particle id and then the index of the next particle in that grid cell. The spatial complexity is O(m+n), where m is the hash table size and n is the number of particles, and the computational complexity is O(n).

With the spatial hash table constructed, a two-dimensional array can be made to store the neighbors for each particle. As most GPU's require the size of two-dimensional lists to be statically allocated, a maximum neighbor count, MN, can be set. To build the neighbor list we can iterate over the 27 immediate and neighboring cells for each particle, compute their hash indices and then traverse the linked list adding particles within the particle's support radius along the way. A neighbor count is also stored for each particle. This process has a spatial and computational complexity of O(MN*n).

C. Constraint Solving

Constraint solving is where we implement the math explained at the beginning of this section. Constraint solving consists of a for loop which runs n iterations of position updates in order to move the particles such that incompressibility has been enforced. There are two procedures that occur during every iteration. The first is that λ_i is computed using equation (8) for all of particles and then stored in an array. Then $\Delta \mathbf{p}_i$ is computed using Equation (11) for each particle. After $\Delta \mathbf{p}_i$ has been calculated $\Delta \mathbf{p}_i$ is altered by collision detection and response to avoid particles getting pushed into objects during the constraint solving process. Collision detection and response will be discussed more in the next section. The altered $\Delta\mathbf{p}_i$ is then stored in an array as well. Finally, the particles positions, $\mathbf{p}_i^*$, are updated.

D. Collision Detection and Response

At the end of the constraint solving, step collision detection and response is performed. In this implementation, bounding boxes with signed distance fields are used to support collision detection and response. Signed distance fields (SDF) are three-dimensional grids that contain the signed distance of each cell to the closest point on the object as well as the corresponding normal vector [9]. Figure 4 shows a cross-section of the signed distance field for a sphere.

By using a grid structure, collision detection and response can be done in O(n*d), where n is the number of particles, and d is the number of objects. At the beginning of the simulation, each of the objects is loaded in, and then their SDF is precomputed using Algorithm 2. The signed distance between the triangles and the cells is calculated using the equation outlined in [7]. The complexity of computing the SDF is $O(w*h*d*t)$, where w, h, and d are the dimensions of the grid, and t is the number of triangles in the object's mesh.

Algorithm 2: SDF Calculation

```
forall cells i do
    minTri ← NULL
    sDist ← $\infty$
    forall triangles j do
        if $|signed\_dist(i,j)| < |sDist|$ then
            minTri ← j
            sDist ← signed_dist(i,j)
        end
    end
    i.dist ← sDist
    i.normal ← minTri.normal
end
```

Once the SDF is computed particle-object collision detection can be performed by first transforming the ith particle's position $\mathbf{p}_i^*$ into grid coordinates $\mathbf{p}_{i,grid}$ using equation (13), which is in the Appendix. Where resolution is the width of a grid cell and minX/Y/Z is the position of the cell corresponding to grid[0][0][0] in world space. Collision response is done by updating $\Delta\mathbf{p}_i$ using equation (14), which is also in the Appendix. Where r is the particle radius and **b** is the dimensions of the SDF grid.

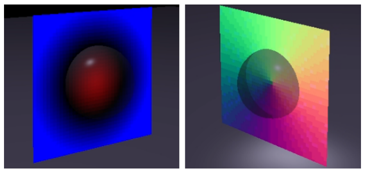


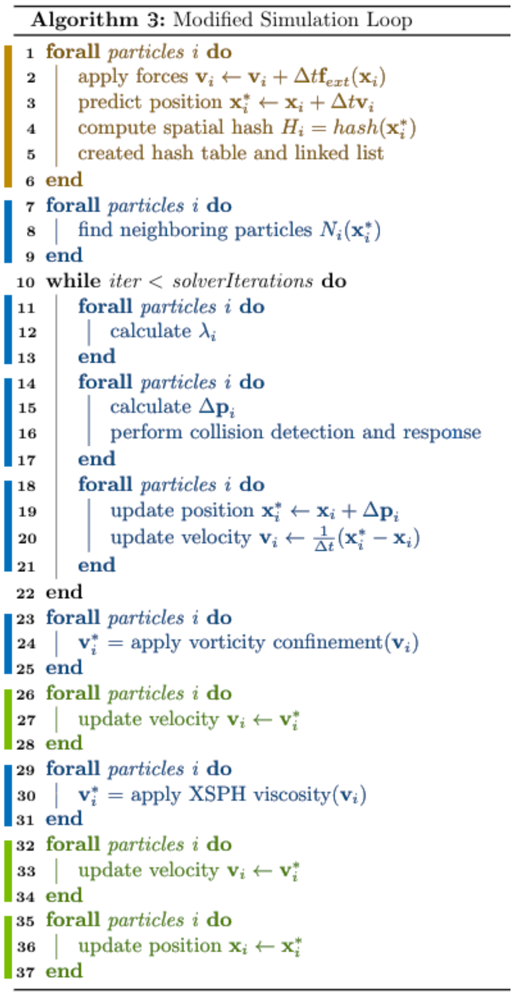
Fig. 4: Sphere Signed Distance Field, Top: signed distance where red is negative distance and blue is positive distance, Bottom: normal vector where (x,y,z) are mapped to (r,g,b)

E. Velocity Update

The final step of the simulation loop is performing the velocity update. The first step is to compute the new velocity of the particles using the equation at line 21 of Algorithm 1. After that, vorticity confinement is applied to replace lost energy, where vorticity is the curl of the velocity. Vorticity confinement calculates the vorticity at the particle's location and then increases the velocity of the particle using the curl [10]. The equations for calculating this force can be found in [4]. Then XSPH viscosity is applied to the velocity which corrects the velocity of the particle to be more similar to its neighbors. Finally, the velocity and position of the particle are updated.

F. OpenGL Implementation

The actual implementation of this simulation was done entirely in OpenGL. Converting the simulation loop into shaders and buffers took quite a few attempts and failures. But finally I implemented the algorithm below:



Algorithm 3: Modified Simulation Loop

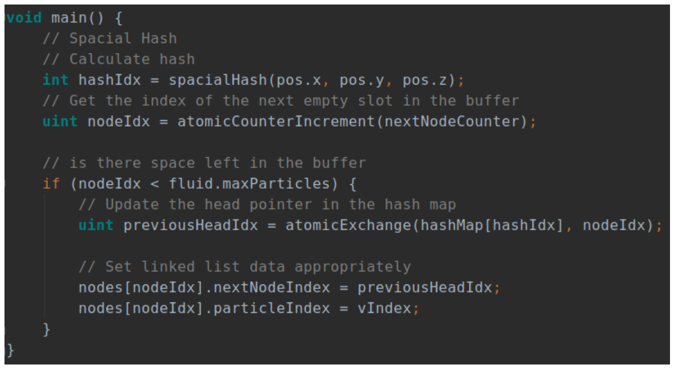
```
forall particles i do
    apply forces $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\mathbf{f}_{ext}(\mathbf{x}_i)$
    predict position $\mathbf{x}_i^* \leftarrow \mathbf{x}_i + \Delta t\mathbf{v}_i$
    compute spatial hash $H_i = hash(\mathbf{x}_i^*)$
    created hash table and linked list
end
forall particles i do
    find neighboring particles $N_i(\mathbf{x}_i^*)$
end
while $iter < solverIterations$ do
    forall particles i do
        calculate $\lambda_i$
    end
    forall particles i do
        calculate $\Delta\mathbf{p}_i$
        perform collision detection and response
    end
    forall particles i do
        update position $\mathbf{x}_i^* \leftarrow \mathbf{x}_i + \Delta\mathbf{p}_i$
        update velocity $\mathbf{v}_i \leftarrow \frac{1}{\Delta t}(\mathbf{x}_i^* - \mathbf{x}_i)$
    end
end
forall particles i do
    $\mathbf{v}_i^*$ = apply vorticity confinement($\mathbf{v}_i$)
end
forall particles i do
    update velocity $\mathbf{v}_i \leftarrow \mathbf{v}_i^*$
end
forall particles i do
    $\mathbf{v}_i^*$ = apply XSPH viscosity($\mathbf{v}_i$)
end
forall particles i do
    update velocity $\mathbf{v}_i \leftarrow \mathbf{v}_i^*$
end
forall particles i do
    update position $\mathbf{x}_i \leftarrow \mathbf{x}_i^*$
end
```

Fig. 5: OpenGL Implementation, mustard is a vertex shader pass, blue is a compute shader pass, and green is a buffer copy

The particle data: $\mathbf{p}$, $\mathbf{p}^*$, $\mathbf{v}$, $\mathbf{v}^*$, λ, $\Delta\mathbf{p}$, and color are all stored in separate Shader Storage Buffer Objects (SSBO). This enables their data to be modified easily by all shaders (mainly Compute and Vertex) while also acting as Vertex Buffer Objects (VBO) for the particle rendering pass. Separate SSBOs were chosen instead of one interlaced SSBO because most of the shaders only use and modify two or three of the attributes. Following suit all of the other fluid simulation data structures also use SSBOs. Those being the spatial hash table and corresponding linked list, the neighbor list, and the SDFs for each object.

The first step to the fluid simulation is to precompute the Signed Distance Field. This is done by first loading in the object's mesh information from Wavefront object files. Then a complete list of triangles and their associated normals are constructed and buffered to the GPU in an SSBO. The SDF's bounding box, transformation matrix, resolution, and dimensions are calculated and buffered to the GPU as well. The compute shader is then dispatched with a workgroup count equal to the grid dimensions, such that each invocation computes the signed distance for each cell. After the SDF has been computed the triangle buffer is unallocated to keep the memory usage manageable.

After that position prediction and spatial hashing are performed simultaneously using a vertex shader with rasterization disabled. The reason that a vertex shader was used to perform this operation is that the method used to compute the spatial hash table requires memory synchronization. The shader implementation of this is shown below:



```
void main() {
    // Spacial Hash
    // Calculate hash
    int hashIdx = spacialHash(pos.x, pos.y, pos.z);
    // Get the index of the next empty slot in the buffer
    uint nodeIdx = atomicCounterIncrement(nextNodeCounter);

    // is there space left in the buffer
    if (nodeIdx < fluid.maxParticles) {
        // Update the head pointer in the hash map
        uint previousHeadIdx = atomicExchange(hashMap[hashIdx], nodeIdx);

        // Set linked list data appropriately
        nodes[nodeIdx].nextNodeIndex = previousHeadIdx;
        nodes[nodeIdx].particleIndex = vIndex;
    }
}
```

The idea is that each invocation of the vertex shader computes the spatial hash of their associated particle. Then we get the first available index in the linked list using the atomic counter increment. And then finally we get the index of the next particle in our cell from the hash table and swap it for our index. This shader is a very simple way to create the spatial hash table but it requires two atomic operations. Compute shaders only supports atomic operations for invocations in the same workgroup, which has a maximum size of one-thousand. As the fluid simulation uses fifteen-thousand or more particles, that won't suffice. So a vertex shader that does support full atomic operations is used. Since the rasterization and fragment shading would take up precious processing time, they were disabled.

The idea of memory synchronization continued to be one of the largest factors in the rest of the implementation's design. Each shader cannot modify data when other invocations may use that data. This leads to five individual compute shader passes which each compute a piece of the equation and then store the results for the next shader to use.

To finish up the neighbor finding step a compute shader pass uses the spatial hash table to construct a list of neighbors for each particle.

With neighbor finding completed, we begin the constraint solving step. A for loop CPU side dispatches the three compute passes which calculate the λs, position update, and then apply the position update.

The last step is to perform the velocity update. A vorticity confinement shader computes the new velocity and then stores those velocities in the $\mathbf{v}^*$ SSBO. Once all the new velocities have been updated an OpenGL CopyBufferData() call is used to efficiently update the velocity buffer. Then an XSPH shader applies XSPH and stores the result as well in the $\mathbf{v}^*$ SSBO. Finally, two buffer copies are performed to update the position and velocity.

The fluid is then rendered as spheres. Hardware instancing is used to perform this operation efficiently. The instanced attributes are the particle's positions to offset the sphere model by and the particle's colors.

V. Results

I tested my algorithm with four different scenarios: a water drop, wave, wave with sphere (Figure 6), and a funnel pour (Figure 7). In all of these scenarios, the fluid behaved properly and object collisions are handled very well.

The goal to create a real-time fluid simulation using a particle system that can interact with complex objects was not achieved. With only fifteen-thousand particles on an NVIDIA GTX 1050TI, we were only able to achieve framerates in the range of 3-5 frames per second. The low framerates come from a few different factors. The first is that I was unable to tune the fluid parameters to support a small support radius. The computational efficiency of this simulation is largely reliant on the number of neighbors each particle has and that number grows in a polynomial fashion as the support radius increases. The second reason is the hash table is constructed in a manner which results in very poor memory locality. Particles within the same cell are strewn across the linked list and this slows down neighbor searching dramatically. Hash table construction also slows down the simulation loop because of the two critical sections. A fully parallelizable hash table construction algorithm could result in a major speedup. And finally, ping-ponging between buffers was not utilized resulting in many large copy operations.

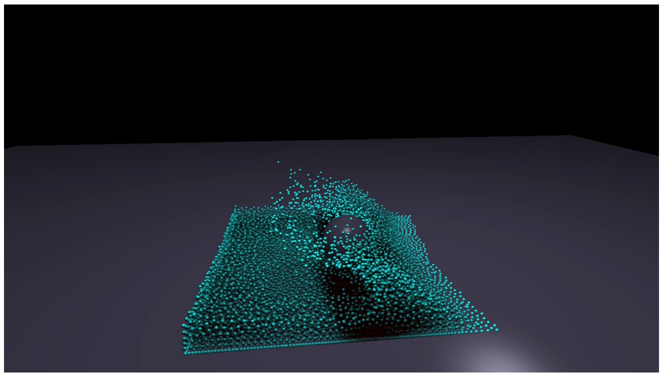


Fig. 6: Colliding a wave with a sphere (15k particles)

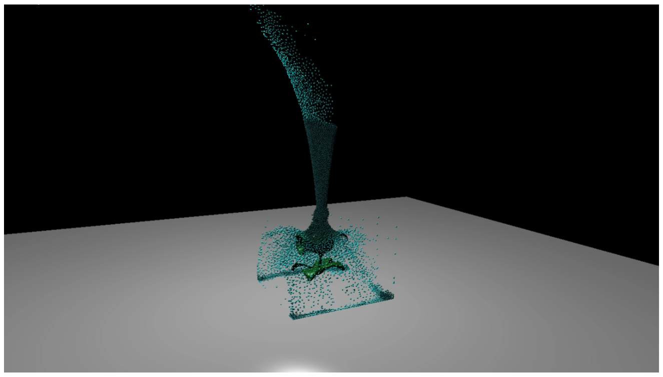


Fig. 7: Pouring water out of a funnel onto a pea shooter (15k particles)

VI. Conclusion

Simulating fluids for real-time graphical applications using particle systems is challenging. However, when done correctly it has impressive results visually. Including simulated fluids in games can produce incredibly immersive gameplay. As modern graphical processors become increasingly powerful these simulations may become commonplace.

For this project, I focused on implementing a real-time fluid simulation based on the work by Miles Macklin and Matthias Müller titles ”Position Based Fluids”. Their solution has been shown to produce comparable accuracy to SPH while being efficient enough for interactive simulations. Implementing their work requires the use of many clever data structures and effective GPGPU.

I was not able to produce the same results as [4], however, the current implementation is a great starting point and with some more work, it could run at similar speeds. This can be done by using a more effective spatial hash table, buffer ping-ponging, and condensing some shader passes.

This project was not without its challenges: parameter tuning, spatial hashing, memory synchronization, and SDF were all incredibly hard to implement correctly and even harder to debug. Through this project, I found that careful planning, pseudocode, research, and color debugging were all very useful techniques when problems arose. In the future, this project could be improved in many ways. After implementing the aforementioned changes, I would like to add in trilinear interpolation of the SDF for improved collision detection and response, fluid rendering techniques to produce realistic-looking liquids, and possibly modify the parameters and structure to simulate smoke or fire. I could also look into adaptive particle sizing, time-stepping, and solver iterations. There were also some projects which explore using a few particles to generate a flow field and then update and render a large number of particles using this field.

Despite not being able to produce real-time results this project was a complete success. I have learned so much more about fluid simulations, OpenGL, computer graphics, and parallel computing. This project was a large step into learning GPGPU and I would love to learn and grow more in this field.

VII. Appendix

$$\mathbf{p}_{i,grid} = \left\lfloor \begin{pmatrix} 1/resolution & 0 & 0 & 0 \\ 0 & 1/resolution & 0 & 0 \\ 0 & 0 & 1/resolution & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -minX \\ 0 & 1 & 0 & -minY \\ 0 & 0 & 1 & -minZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{p}_i^* \right\rfloor \tag{13}$$

$$\Delta\mathbf{p}_i = \begin{cases} \Delta\mathbf{p}_i + (r - grid[\mathbf{p}_{i,grid}].dist) * grid[\mathbf{p}_{i,grid}].norm & \text{if } \mathbf{0} \le \mathbf{p}_{i,grid} \le \mathbf{b} \wedge grid[\mathbf{p}_{i,grid}].dist \le r \\ \Delta\mathbf{p}_i & \text{else} \end{cases} \tag{14}$$

References

[1] N. Hall, “Navier-Stokes Equations,” NASA, 05-May-2015. [Online]. Available: https://www.grc.nasa.gov/WWW/k-12/airplane/nseqs.html. [Accessed: 26-Apr-2020].

[2] Kellomäki, Timo. “Large-Scale Water Simulation in Games,” Tampere University of Technology, 2015. 91 p. (Tampere University of Technology. Publication; Vol. 1354).

[3] J. Monhagan , “Smoothed particle hydrodynamics,” Annual Review of Astronomy and Astrophysics 30, 1, 543–574, 1992.

[4] M. Macklin and M. Müller, “Position Based Fluids,” ACM Transactions on Graphics, vol. 32, no. 4, p. 1, Jan. 2013.

[5] Müller M., Charypar D. and Gross M., 2003. Particle-based fluid simulation for interactive applications. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA ’03, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association, 154–159.

[6] A. Wongpanich and O. Jow, “Position-Based Fluid Simulation,” OwenJow. [Online]. Available: https://owenjow.xyz/graphics/1845/. [Accessed: 28-Apr-2020].

[7] J. Monhagan. ”SPH Without a Tensile Instability,” Comput. Phys. 159, 2 (Apr.), 290–311, 2000.

[8] Bilotta, Giuseppe, Vito Zago and Alexis Hérault. “Design and Implementation of Particle Systems for Meshfree Methods with High Performance.” (2018).

[9] M. Mroz, “Signed Distance Fields in Real-Time Rendering,” Apr. 2017.

[10] J. Rampe, “Vorticity Confinement for Eulerian Fluid Simulations,” Softology, 15-Nov-2019.

[11] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner, “SPH Fluids in Computer Graphics,” EUROGRAPHICS, 2014.