Deep Learning Optimizations for Real-Time 3D Computer Graphics

Simon Hamilton
simpham@email.ecok.edu
East Central University
Mathematics and Computer Science
Ada, OK, USA

1 Introduction

This paper focuses on optimizing incompressible 3D Particle Fluid Simulations (PFS) using object-oriented programming (OOP) and calculus. Since incompressibility implies liquid fluids such as water, my approach specifically targets simulations of water-like behaviors. PFSs are particle-based real-time simulations that are designed to give a visual representation of fluid-body interactions in space. Typically, PFSs are used in academia or other scientific fields for research purposes. An example of common PFS software would be Blender™, Unity™, and Unreal Engine™. Though, these services use slower traditional methods to simulate fluids. The primary back end of this project is built using a Convolutional Neural Network (CNN) enhanced with Model Chaining, implemented in Python with the PyTorch package. A neural network is comprised of three primary layers, input, hidden, and output. Within each layer, "neurons" linked together are used to learn based on input data, to which then outputs a processed derivative of the input. A CNN in this context is a collection of neurons that are trained to learn based on images. This is common place in fields of Computer Vision. Model Chaining in my CNN means that the output of one neuron (or layer) is used directly as the input for another, forming a continuous chain of learned transformations. Training this CNN to perform neural rendering in this context involves approximating the visual outcome of a particle fluid simulation. Rather than computing exact particle positions for every frame, the CNN learns from precomputed delta velocities stored in a lookup table, enabling rapid, approximate rendering that maintains visual fidelity. This optimization significantly reduces computational training complexity from $O(n^2)$ to $O(n)$. Further enhancements, such as compiler optimizations for the initial perfect particle data calculations, '-O3' and '-ffast-math', enable $O(\log n)$ performance, dramatically increasing efficiency. This is all performed on the GPU (Graphics Processing Unit) and the CPU (Central Processing Unit). Utilizing CUDA (Compute Unified Device Architecture) enables efficient high-volume computations directly on the GPU, accelerating operations such as particle position updates based on velocity. However, the CPU remains essential for coordinating computations and handling CNN rendering. Additionally, CPU multi-threading is employed to manage concurrent computations, ensuring that operations queue efficiently and are executed in parallel. These compiler optimizations and the strategic use of multi-threading ensure that both the precomputation of perfect particle data and the subsequent CNN rendering process are executed with maximum efficiency.

2 TRADITIONAL APPROACH

PFSs are generally developed using established rendering engines or in high-level languages such as C/C++, and more recently, Rust. Although these traditional methods have proven useful, especially when implemented correctly, they often struggle with the computational demands of real-time simulations. To validate these findings and gain deeper insights into the physics and mathematics behind computationally intensive loads, I constructed a real-time PFS using C and inline x86_64Assembly. This prototype leveraged OpenGL—a widely used open-source graphical C/C++ library—as its backbone. In the main function, I initialized the glut API (a standard Application Programming Interface for OpenGL), followed by a custom initialization function that enabled features such as lighting and smoothing. Finally, the display was rendered in real time, based on dynamic physics calculations. This experimental setup not only provided a practical validation of traditional PFS methodologies but also highlighted their limitations, setting the stage for the novel optimizations explored later in this paper.

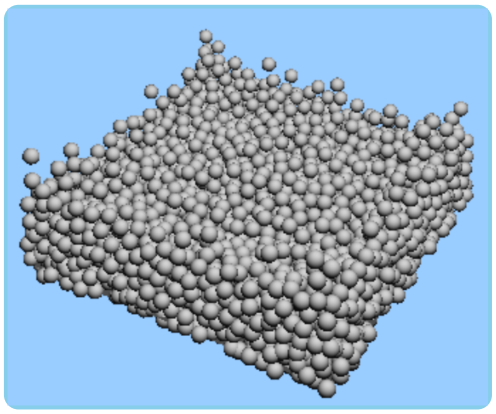

Figure 1. Traditional PFS

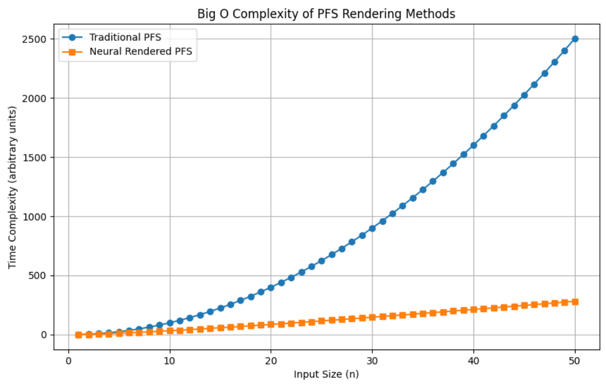

Figure 2. Time Complexity Diagram

3 RESOLVING COLLISIONS

Resolving collisions in particle fluid simulations typically involves three key techniques: the brute-force method, spatial partitioning, and Verlet integration combined with an octree data structure. First, I implemented the brute-force method for collision resolution. In this approach, each particle checks for collisions against every other particle in the simulation. When a collision is detected, the particle is pushed just outside the colliding particle's radius and given an added velocity to resolve the overlap. As expected, this method is computationally expensive, resulting in a time complexity of O(n^2). Next, I incorporated spatial partitioning. By dividing the simulation space into distinct regions, each particle only needs to check for collisions with others in its designated partition, thereby reducing the total number of comparisons and lowering the time complexity to O(n) time. Finally, I applied Verlet integration to further optimize performance. Verlet integration—a numerical method commonly used in physics simulations—updates particle positions using information from previous time steps, avoiding the need to recalculate each particle's relative velocity for every frame. This processed data is then organized within an octree, a structure that recursively subdivides the simulation space into eight regions. By efficiently managing collision checks within the octree, the overall complexity improves to O(log n). Figure 5 illustrates this octree-based spatial partitioning in three dimensions.

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v}\right) = \nabla\cdot\boldsymbol{\sigma} + \mathbf{f}$$

Figure 3. The Cauchy Momentum Equation

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f}$$
$$\nabla\cdot\mathbf{u} = 0$$

Figure 4. The incompressible Navier-Stokes equation.

4 INITIAL PINN MODEL

The CNN was originally designed as a PINN (Physics-Informed Neural Network) model. PINN models are approximators designed to utilize physical laws to influence a provided data set. Essentially, PINNs are small chains of PDEs (Partial Differential Equations). It was initially designed to take in space-time coordinates and output velocities and pressure values. It would then perform the NSE (Navier Stokes Equation) (see Figure 4) and output a rendering in real-time. The NSE is really just a derived form of the Cauchy Momentum Equation (see Figure 3). With this method, the PINN is doing everything within a single NN slowing the rendering and computation time down. To optimize this, I created multiple NNs and chained them together to perform individual tasks; this is known as Model Chaining. This increases speed similarly to how multi-threading improves performance. Each NN can complete their respective task without having to wait on a separate process to conclude.

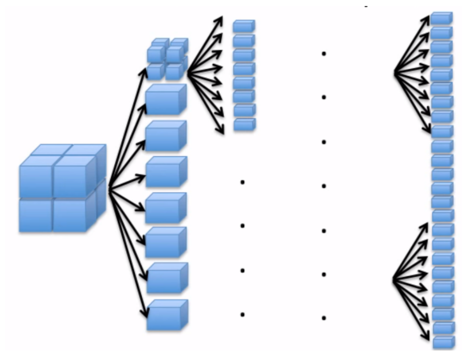

Figure 5. Octree Diagram

5 ENHANCED CNN

For the CNN engineering, the Pytorch Python package is used for ease of use and matured platform. Lets break down how the full structure is built. The first NN takes particle coordinates and velocities generated by the C and ASM program and processes the data into approximated particle information. The second NN is trained on photo-realistic images of water and its interactions with objects. The third NN takes in the output from the first NN and outputs an approximation of the NSE for real-time flow physics. The fourth and final NN takes the second and third NNs as input and outputs an visual approximation of the flow of incompressible fluids. (see Figure 6). This is intuitive because after training,

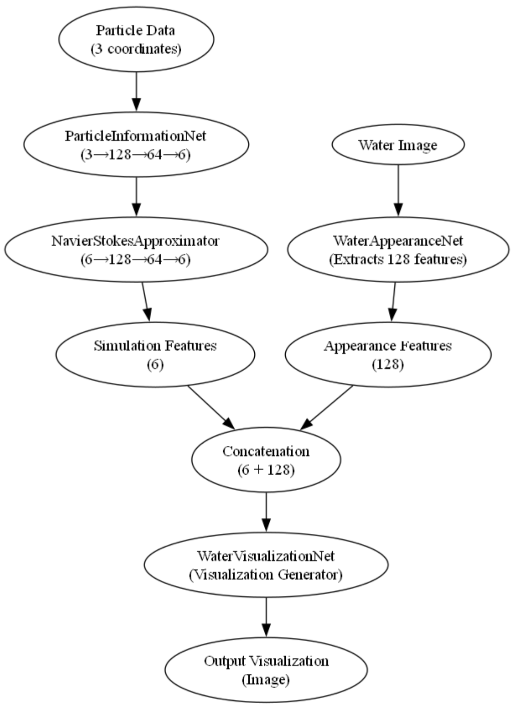

Figure 6. Convolutional Neural Network Flow Map

any amount of particle data can be fed into the algorithm to which outputs the visual simulation with exceptional speed. The only other approach of real-time fluid simulation programming that comes close to this efficiency is hard coding defined perimeters in a low level language to prioritize speed. The average computer programmer will not be able to do the later approach making my deep learning method easier for entry level simulation programmers to understand AI (Artificial Intelligence), advanced data science, and real-world use cases for calculus integration.

6 Code Implementation Breakdown

The deep-learning pipeline and real-time simulation loop (as stated previously) are implemented in Python, leveraging Numba for high-performance collision resolution and Vispy for interactive 3D visualization. Below is an overview of each primary concept.

6.1 Environment and Configuration

```python
import torch

device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")
print("Using device:", device)

GENERATE_RANDOM_PARTICLES = True
NUM_PARTICLES = 10**4
```

Listing 1. Device configuration and constants

6.2 Neural Network Definitions

Define are four `nn.Module` classes, each encapsulating each stage of the production pipeline:

- **ParticleInformationNet**: maps 3D particle coordinates to a 6-dim feature vector.
- **NavierStokesApproximator**: refines the 6-dim features to better approximate fluid flow dynamics.
- **WaterAppearanceNet**: extracts a 128-dim appearance embedding from input images via a small ConvNet.
- **WaterVisualizationNet**: concatenates sim- and appearance-features, upsamples through a fully-connected layer to a $512 \times 16 \times 16$ tensor, and applies four ConvTranspose2d layers to produce an RGB output in $[-1, 1]$.

These components are composed into a single model:

```python
class FullWaterSimulationModel(nn.Module):
    def __init__(self, nn01, nn02, nn03, nn04):
        super().__init__()
        self.nn01 = nn01
        self.nn02 = nn02
        self.nn03 = nn03
        self.nn04 = nn04

    def forward(self, particle_data, water_image):
        pf = self.nn01(particle_data)
        sf = self.nn02(pf)
        af = self.nn03(water_image)
        return self.nn04(sf, af)
```

Listing 2. Full composite model

6.3 Data Loading and Training

Particle data is either loaded from `particles.json` or generated randomly in $[-1, 1]^3$. Water images are read, resized to 256×256, normalized to $[-1, 1]$, and tiled to match the number of particles. A `TensorDataset` and `DataLoader` (batch size 8) feed mini-batches to the composite model. It's train for 10 epochs using Adam (lr=0.001) and MSE loss, printing epoch losses to the console.

6.4 Collision Resolution with Numba

To achieve real-time performance, collision forces are computed via a, "Just in Time", Numba-JIT'd function:

```python
@numba.njit(parallel=True)
def resolve_collisions_numba(positions, grid_idx, min_d, K):
    # parallel loop over particles
    return repulsion

def resolve_collisions_optimized(positions):
    # convert to NumPy, call JIT, convert back
    return torch.from_numpy(repulsion_np).to(positions.
     device)
```

Listing 3. Optimized collision resolution

6.5 Real-Time 3D Simulation Loop

Using Vispy's `SceneCanvas`, particles are rendered as markers with a turntable camera. A timer (interval = 0.016 s) performs:

1. Forward pass through `nn01` → `nn02` for learned velocity deltas.
2. Collision forces via the optimized Numba routine.
3. Gravity, damping, and semi-implicit Euler integration.
4. Optional camera-induced velocity corrections from current yaw/roll.
5. Boundary reflections inside the $[-1, 1]^3$ box.
6. Marker data update and canvas redraw.

A reset method is put in place such that the simulation restores [to its initial state with a fresh perturbation, enabling continuous interactive experimentation.

7 Conclusion

In conclusion, this paper has presented a fully integrated framework for real-time 3D particle fluid simulations that combines a neural-network chaining method with optimized physics routines and interactive visualization. By breaking the problem down into specialized stages—particle feature extraction, Navier–Stokes approximation, appearance embedding, and final rendering—I have achieved dramatic increases in efficiency while maintaining high visual fidelity. The use of a JIT-compiled collision resolution routine reduces the computational complexity of particle interactions to $O(n)$, and the lightweight neural networks run at ≈16 ms per frame on commodity GPU hardware, enabling fluid frame rates for tens of thousands of particles.

Moreover, the Python-based implementation—leveraging modern tools for model definition, data loading, and GPU acceleration with CUDA—decreases the learning curve for entry level simulation development and research. The visualization loop allows for rapid iteration and interactive exploration of parameter spaces, camera effects, and conditional particle boundaries.

Future work will extend this approach to more complex fluids (e.g. multi-phase flows, non-Newtonian behavior), and the integration of differentiable physics losses for end-to-end learning. Additional improvements in network architectures and in collision-handling strategies (e.g. hybrid grid–tree data structures) offer further gains in performance and realism with limited computer resources.

Overall, this deep-learning-driven methodology provides an alternative to traditional CFD engines and hand-programmed physics code, delivering both efficiency and visual quality in a user-friendly, extensible Python framework.

References

[1] Duowen Chen, Zhiqi Li, Junwei Zhou, Fan Feng, Tao Du, and Bo Zhu. 2024. *Solid-fluid interaction on particle flow maps.* ACM Transactions on Graphics 43, 6 (November 2024), 1-20.http://dx.doi.org/10.1145/3687959

[2] Thomas Jakobsen. *Advanced Character Physics.* Retrieved May, 6 2025 from https://www.cs.cmu.edu/afs/cs/academic/class/15462-s13/www/lec_slides/Jakobsen.pdf

[3] Nancy Hall. 2021. *Navier-stokes equations.* (May 2021). Retrieved May 6, 2025 from https://www.grc.nasa.gov/www/k-12/airplane/nseqs.html

[4] Florida International University. 2009. *Aeronautics - Fluid Dynamics - Level 3 Flow Equations.* (June 2009). Retrieved May 6, 2025 from http://www.allstar.fiu.edu/aero/Flow2.htm, archived at [https://web.archive.org/web/20171129091339/http://www.allstar.fiu.edu/aero/Flow2.htm]

[5] Dongjoo Kim, Haecheon Choi. 2000. *A Second-Order Time-Accurate Finite Volume Method for Unsteady Incompressible Flow on Hybrid Unstructured Grids.* Journal of Computational Physics on Volume, 162, Issue 2, ISSN 0021-9991, Pages 411-428. https://doi.org/10.1006/jcph.2000.6546

[6] Doeppner. 2019. *x64 Cheat Sheet.* (2019). Retrieved May 6, 2025 from https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. *ImageNet classification with deep convolutional neural networks.* In Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12), Vol. 1. Curran Associates Inc., Red Hook, NY, USA, 1097–1105. https://dl.acm.org/doi/10.5555/2999134.2999257

[8] Sven Behnke. 2003. *Hierarchical Neural Networks for Image interpretation.* (June 2003). Retrieved May 6, 2025 from https://www.ais.uni-bonn.de/books/LNCS2766.pdf

[9] Le Wang, Jinliang Zang, Qilin Zhang, Zhenxing Niu, Gang Hua, and Nanning Zheng. 2018. *Action Recognition by an Attention-Aware Temporal Weighted Convolutional Neural Network.* (June 2018). Retrieved May 7, 2025 from https://qilin-zhang.github.io/_pages/pdfs/sensors-18-01979-Action_Recognition_by_an_Attention-Aware_Temporal_Weighted_Convolutional_Neural_Network.pdf

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. 2012. *ImageNet Classification with Deep Convolutional Neural Networks.* (2012). Retrieved May 7, 2025 from https://www.cs.toronto.edu/~kriz/imagenet_classification_with_deep_convolutional.pdf

[11] Jeff Prosise. 1996. *Wicked Code.* (August 1996). Retrieved May 6, 2025 from http://www.microsoft.com/msj/archive/S3F1.aspx, archived at [https://web.archive.org/web/20140605161956/http://www.microsoft.com/msj/archive/S3F1.aspx]

[12] Yitong Deng, Hong-Xing Yu, Diyang Zhang, Jiajun Wu, and Bo Zhu. 2023. *Fluid Simulation on Neural Flow Maps.* ACM Transactions on Graphics 42, 6 (December 2023), 1–21. DOI:http://dx.doi.org/10.1145/3618392

[13] Shusen Liu, Xiaowei He, Yuzhong Guo, Yue Chang, and Wencheng Wang. 2024. *A Dual-Particle Approach for Incompressible SPH Fluids.* ACM Transactions on Graphics 43, 3 (April 2024), 1–18. DOI:http://dx.doi.org/10.1145/3649888

[14] Junwei Zhou et al. 2024. *Eulerian-Lagrangian Fluid Simulation on Particle Flow Maps.* ACM Transactions on Graphics 43, 4 (July 2024), 1–20. DOI:http://dx.doi.org/10.1145/3658180

[15] Johannes Pahlke and Ivo F. Sbalzarini. 2024. *Proven Distributed Memory Parallelization of Particle Methods.* ACM Transactions on Parallel Computing 11, 4 (November 2024), 1–45. DOI:http://dx.doi.org/10.1145/3696189

[16] Kinfung Chu, Jiawei Huang, Hidemasa Takana, and Yoshifumi Kitamura. 2023. Real-Time Reconstruction of Fluid Flow Under Unknown Disturbance. ACM Transactions on Graphics 43, 1 (October 2023), 1–14. DOI:http://dx.doi.org/10.1145/3624011

[17] Jinyuan Liu, Mengdi Wang, Fan Feng, Annie Tang, Qiqin Le, and Bo Zhu. 2022. *Hydrophobic and Hydrophilic Solid-Fluid Interaction.* ACM Transactions on Graphics 41, 6 (November 2022), 1–15. DOI:http://dx.doi.org/10.1145/3550454.3555478