Divergence-Free Smoothed Particle Hydrodynamics in a Stream Digital Twin

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Computer Science
Computer Science

---

by
Austin Hartley
May 2025

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Daljit Dhillon
Dr. Matias Volonte

Abstract

Digital Twins (DT) are being explored by the South Carolina (SC) water community to simulate how SC streams will flow at various water levels. Currently, a DT called Gilligan simulates these streams utilizing weakly-incompressible Smoothed Particle Hydrodynamics (SPH). This method does not strictly enforce incompressibility, which leads to unrealistic water flows and unwanted visual artifacts that require post-processing effects to hide. To address these problems and simulate more realistic water flows, the Gilligan stream logic is updated and a state-of-the-art SPH method that enforces incompressibility—Divergence-Free SPH (DFSPH)—is implemented within the Gilligan framework. DFSPH is able to make use of two pressure solvers, one to ensure the fluid stays at constant density and another to keep the velocity field divergence free. Simulations are tested using a model of Hunnicutt Creek on the campus of Clemson University, showcasing the DFSPH water flows as being more realistic, detailed, and not requiring the same visual alterations to hide artifacts compared to the weakly-incompressible SPH.

Acknowledgments

Thank you to my advisor Dr. Tessendorf and my committee members, Dr. Volonte and Dr. Dhillon.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

In recent years, Digital Twins (DT) have become increasingly popular within industry and research communities. DTs are software that realistically simulate real-life phenomena, allowing for analysis and visualizations that may not be plausible to achieve in real life. Creating these simulations are cost-effective and have the additional benefit of being able to be directly controlled by the user to create their desired event. DTs are often being used to create safe training environments, predict natural disasters, or create informative visualizations for the mass public. The South Carolina (SC) water community is constantly looking for ways to improve analyses and create meaningful visualizations. One of their main goals is to educate the public through visualizations of how streams behave at different water levels, such as in a flooding event. Creating a visualization in a real-world location familiar to the intended audience is often more compelling for motivating preventative action than presenting predictive data alone. The SC water community holds an annual conference to give light to research that can further improve their goals. In 2022 a DT called Gilligan [19], showcased the potential of creating a realistic stream visualizations using Hunnicutt Creek located on the campus of Clemson University [20]. The textures, sky, and triangular mesh of the Hunnicutt Creek streambed (Figure 1.1) were captured via on-site images to create a virtual replica of the environment [20]. The SC water community was impressed by the realism of the renders, but wanted a more refined fluid simulation to achieve a more realistic fluid flow.

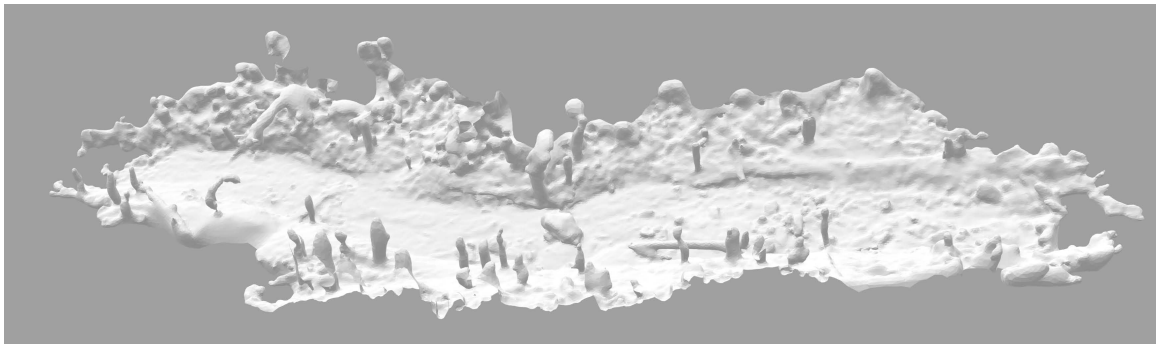

Figure 1.1: Triangular mesh of the Hunnicutt Creek [20].

1.1 Gilligan

Gilligan was originally developed by Dr. Jerry Tessendorf, using an amalgamation of computer graphic techniques to simulate real-life scenarios such as clouds, oceans, mirages, and streams. A few of which can be seen in Figure 1.2 and 1.3 [19]. Gilligan has a back end implemented in C++ with modules for physically-based animation (PBA), volumetric rendering and manipulation, and Global Illumination (GI) path tracing [17], which is accessed through Python scripting to create and visualize different types of simulations. This setup easily allows for the creation of stream simulations to fill any streambed geometry with particles and simulate multiple types of flows.

1.1.1 Navier-Stokes Equations

The first step to mathematically explaining how Gilligan models fluid dynamics is to denote how the fluid's density (mass) changes over time through the continuity equation

$$\frac{D\rho}{Dt} = -\rho(\nabla \cdot \mathbf{v}), \tag{1.1}$$

where $D(\rho)/Dt$ is the material derivative of density with respect to time, ρ is density, and $\nabla \cdot \mathbf{v}$ is the divergence of the velocity. Because water is an incompressible fluid, the continuity equation gets constrained and its momentum conservation is expressed through the isothermal Navier-Stokes equations,

$$\frac{D\rho}{Dt} = 0 \Leftrightarrow \nabla \cdot \mathbf{v} = 0 \tag{1.2}$$

$$\frac{D\mathbf{v}}{Dt} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{v} + \frac{\mathbf{f}}{\rho}, \tag{1.3}$$

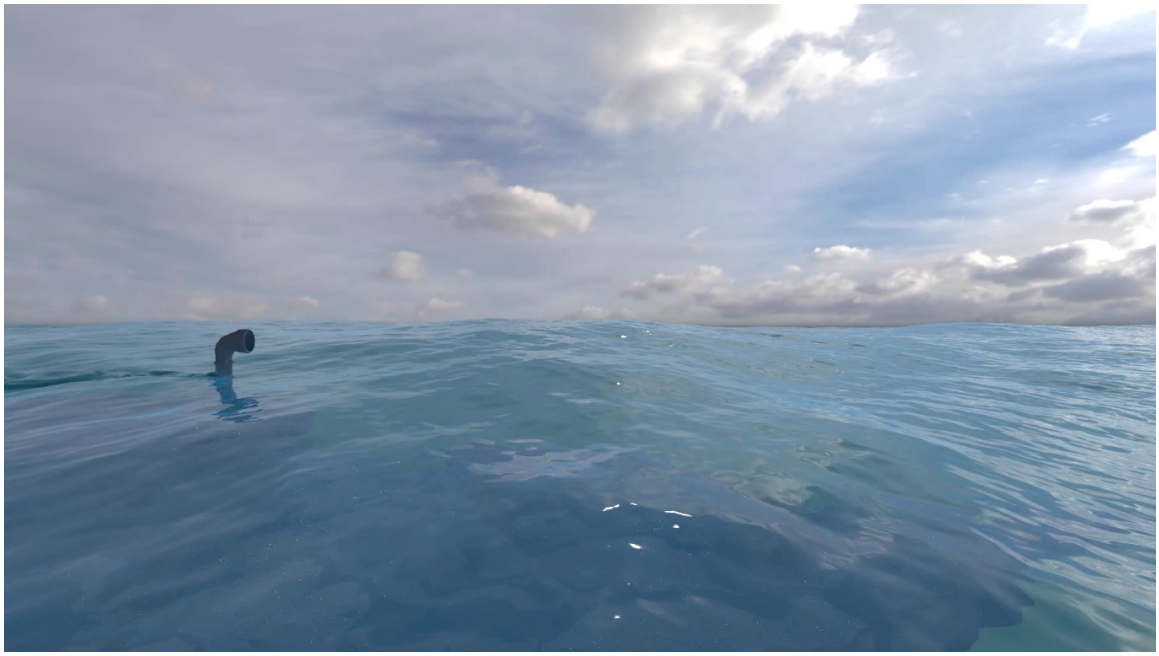

Figure 1.2: Scene of a submarine on the ocean surface during a cloudy atmosphere rendered by Gilligan [19].

where $D(\cdot)/Dt$ is the material derivative, and ρ, $\mathbf{v}$, p, ν, and $\mathbf{f}$ denote density, velocity, pressure, kinematic viscosity, and external forces, respectively. Equation (1.2) showcases that an incompressible fluid must have a density derivative with respect to time equal to zero, which is directly related to the velocity field being divergence free. If a fluid is not losing or gaining density over time, it is impossible for it to expand or collapse within its defined boundaries. In equation (1.3), the fluid is being advected with respect to time based off its velocity. The fluid's velocity is composed of its dynamic viscosity (kinematic viscosity coefficient multiplied with the Laplacian of its current velocity), external forces such as gravity, and its pressure gradient. Incompressible fluids move from high pressure to low pressure, which is why the negation of the pressure gradient is used to reflect the fluid in the correct direction. This dynamic ensures it stays incompressible, so accurately solving for pressure is vital and one of the most complex aspects of fluid simulations. There is no general analytical solution for the Navier-Stokes equations, and one of the most popular methods to discretize these equations is through Smoothed Particle Hydrodynamics (SPH), which makes use of operator splitting to solve this complex Partial Differential Equation (PDE) in smaller steps.

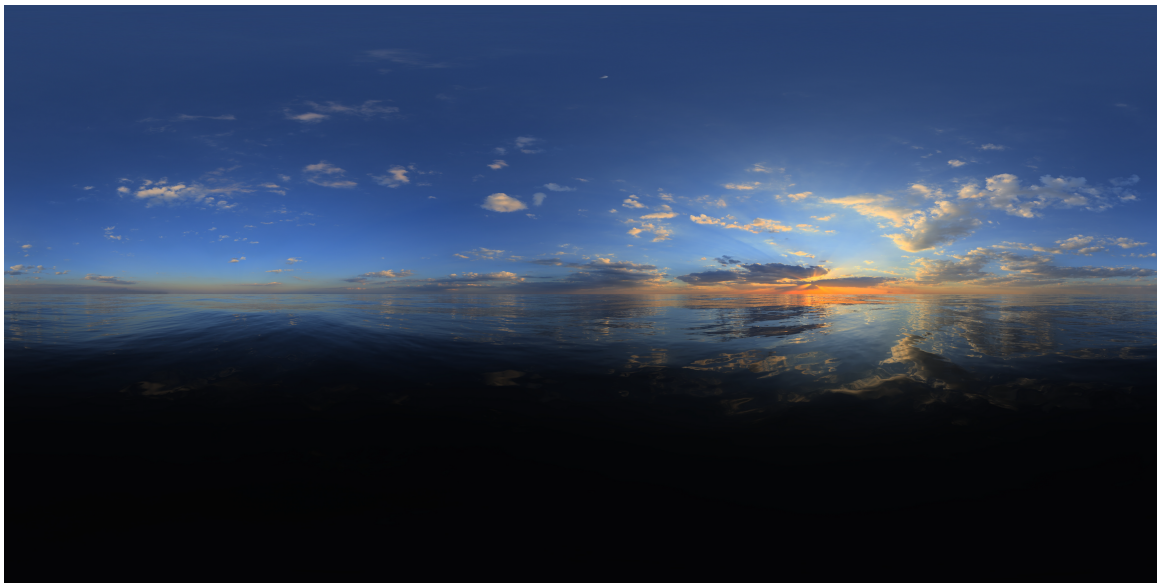

Figure 1.3: Scene of a colorful sky over an ocean surface rendered by Gilligan [19].

1.2 Fluid Simulation in Gilligan

Gilligan's original stream simulations were promising, but did not capture the realism of a natural stream or the needs of the SC water community. This is because its PBA module used a simple weakly-incompressible SPH solver to model fluids as particles. Water, being an incompressible fluid, is poorly modeled in this way, so the method created unrealistic water simulation data and as a result unwanted artifacts in the visualization. The weakly-incompressible SPH method inaccurately solves for pressure using the Tait Equation of State (EOS) [11], which does not strictly enforce the constraints in equation (1.1) [19]. Figures 1.5 and 1.4 contain visualizations of the weakly-incompressible method that showcase particles creating unrealistic surfaces due to pressure errors, which is more noticeable with higher particle counts. This problem, along with the unrealistic water flow, is even more obvious in motion.

To combat this, a more sophisticated solver called Divergence-Free SPH (DFSPH) [3] will be implemented that accurately solves for incompressibility as well as additional viscosity and time step improvements. The DFSPH method accurately adheres to those constraints and creates improved water flows and surfaces. DFSPH uses two pressure solvers, one to keep constant density and the other to enforce a divergence-free velocity field within a user-defined threshold [3]. The first stream simulations in Gilligan used artificial viscosity, introduced by Monaghan and Gingold in

Figure 1.4: Water surface visualization of the weakly-incompressible SPH stream simulation at a high water level (11 million particles) [20].

1983 [16], which has been updated to an explicit viscosity equation approximating the Laplacian of the current velocity introduced by Monaghan in 2005 [15]. The new solver utilizes a dynamic time step to improve stabilization, which is an additive for better simulation data. The current DFSPH simulations accurately simulate fluid, but suffer from flyaway particles, which is a common problem in SPH. With the new DFSPH solver, the rendering has the same pipeline as before, utilizing Gilligan's built-in functionality to create a gridded volume of the fluid using the Signed Distance Function (SDF) for spheres in the location of the particles. The previous SPH method required blurring during volume creation to mitigate visual artifacts on the water's surface, which created viable stand-still photos at lower water levels as seen in Figure 1.6. This post-processing effect is no longer necessary with the increased accuracy of the DFSPH solver. Volumes can be converted into geometry via Marching Tetrahedra (MT) [4], and GI path tracing is used for geometrically realistic lighting in a scene, allowing different materials to have distinct shaders [19]. To address the flyaway particles, a new shader is implemented in the render module to cull any particles that do not meet the user-specified minimum particle count in their vicinity during rendering. The shader does not completely resolve the problem as culled particles close to the geometry still cast a shadow.

Figure 1.5: Particle visualization of the weakly-incompressible SPH stream simulation at a medium water level rendered using Gilligan's lambertian shader [20].

Figure 1.6: GI render of the volume converted to the water surface geometry for the weakly-incompressible SPH stream simulation at a medium water level [20].

Chapter 2

Related Work

Smoothed Particle Hydrodynamics (SPH) was developed by astrophysicists in 1977 because their phenomena were not efficiently modeled by grid-based Eulerian methods [5]. SPH is a Lagrangian method where phenomena are modeled by individual particles and attributes are tracked through discrete time steps. This method has been adopted in many engineering fields [14] and is especially used in computer graphics [8] to simulate fluids, with water being one of the most common.

2.1 Viscosity

One part of solving for the velocity of a fluid, according to equation (1.3), is accounting for its viscosity. One of the simplest ways to solve for this term is through artificial viscosity [16], written as

$$\Pi_{ij} = \begin{cases} \frac{-\alpha c_{\bar{ij}} \mu_{ij} + \beta \mu_{ij}^2}{\rho_{\bar{ij}}}, & \text{if } \mathbf{r}_{ij} \cdot \mathbf{v}_{ij} > 0 \\ 0, & \text{if } \mathbf{r}_{ij} \cdot \mathbf{v}_{ij} \leq 0 \end{cases}, \tag{2.1}$$

where r_{ij} is the difference between the positions of particle i and j, v_{ij} is the difference between the velocities of particles i and j, $c_{\bar{ij}}$ is the average speed of sound between the two particles, α and β is user input (usually between 1 and 2) [1], and $\rho_{\bar{ij}}$ is $(\rho_i + \rho_j/2)$. The term μ_{ij} is defined as [16]

$$\mu_{ij} = \frac{\mathbf{r}_{ij} \cdot \mathbf{v}_{ij}}{h\left(\|\mathbf{r}_{ij}\|^2/h^2 + \eta^2\right)}, \tag{2.2}$$

where h is the smoothing kernel radius and η is user input which is usually set to 0.01 to avoid divergence errors [1]. While this derivation can relieve particle instabilities and produce a viscous-like flow, it is not an accurate representation of the real viscosity term based on the Laplacian of the velocity.

2.2 Incompressibility

Water is an incompressible fluid, and a simple method for enforcing incompressibility is to solve an Equation of State (EOS) [11]. The Tait EOS is one of the most popular, where the pressure, p_i, for each particle is denoted by [11]

$$p_i = \frac{\kappa\rho_0}{\gamma}\left(\left(\frac{\rho_i}{\rho_0}\right)^{\gamma} - 1\right), \tag{2.3}$$

with κ and γ as user stiffness parameters and ρ_0 as the rest density. A simplified version of this is often used in computer graphics which sets γ to one [7],

$$p_i = \kappa(\rho_i - \rho_0). \tag{2.4}$$

More complex methods solve Pressure Poisson Equations (PPE), such as Predictive-Corrective Incompressible SPH (PCISPH), Implicit Incompressible SPH (IISPH), and DFSPH [11]. Solving an EOS is less computationally expensive and accurate than a PPE, which is why it is used in weakly-incompressible simulations and those without accuracy requirements. For the EOS to achieve higher levels of accuracy, a considerably small time step must be used, resulting in a serious performance bottleneck [11].

2.2.1 Pressure Poisson Equations

In order to improve accuracy and allow for higher time steps to increase performance, discretizing the Pressure Poisson Equations (PPEs) to solve for the Navier-Stokes equations in new ways was researched. These methods typically involve solving a linear system with a relaxed Jacobi scheme or one of similar fashion [11]. In solving PPEs, the first step is to predict the velocity based on non-pressure acceleration, $\mathbf{a}_{\text{nonp}}$, written as $\mathbf{v}_i^* = \mathbf{v}_i + \Delta t\mathbf{a}_{\text{nonp}}$, where Δt is the time step [11]. The predicted density, ρ^*, can be found using the divergence of the predicted velocity, $\mathbf{v}^*$, which is

derived through the continuity equation (1.1), and denoted as [10]

$$\rho^* = \rho(t) - \Delta t \rho(t) \nabla \cdot \mathbf{v}^*. \tag{2.5}$$

This is simply integrating the density deviation over time, $\frac{D\rho}{Dt}$, and adding it to the current density.

The point of calculating each particle's pressure is to update their velocity for the next time step [10],

$$\mathbf{v}(t + \Delta t) = -\Delta t \frac{1}{\rho(t)} \nabla p(t) + \mathbf{v}^*, \tag{2.6}$$

based on acceleration due to pressure alongside the non-pressure accelerations. Adhering to equation (1.1), density deviation over time can be denoted as [10]

$$\frac{D\rho}{Dt} = -\rho(t) \nabla \cdot \left(\Delta t \frac{1}{\rho(t)} \nabla p(t) \right), \tag{2.7}$$

where $\mathbf{v}$ from equation (1.1) is filled in as the velocity due to pressure. The predicted density deviation over time, $\frac{\rho_0 - \rho^*}{\Delta t}$, where ρ_0 is the rest density, should negate $\frac{D\rho}{Dt}$, resulting in $\frac{D\rho}{Dt} = 0$ from equation (1.2) [10]. Written out mathematically, this becomes $\frac{\rho_0 - \rho^*}{\Delta t} - \rho(t) \nabla \cdot \left(\Delta t \frac{1}{\rho(t)} \nabla p(t) \right) = 0$ [10]. Pressure can be solved by moving its terms to one side and simplifying the new equation's results in

$$\Delta t \nabla^2 p(t) = \frac{\rho_0 - \rho^*}{\Delta t}, \tag{2.8}$$

which is one of many variations of the PPE equation for density deviation, where the right hand side is referred to as the source term [10]. Solving for pressure results in a linear system based on the number of particles, n, with n equations and n unknowns. The PPE equation for divergence starts from the viewpoint of rearranging equation (2.6), comparing the predicted velocity with its current velocity to find the correct velocity from pressure acceleration, $\mathbf{v}^* - \mathbf{v}(t + \Delta t) = \Delta t \frac{1}{\rho(t)} \nabla p(t)$ [10]. The constraint from equation (1.2) states the divergence of the velocity should be 0. This is written as $\nabla \cdot \mathbf{v}^* = \nabla \cdot \left(\Delta t \frac{1}{\rho(t)} \nabla p(t) \right)$ [10], where the divergence of the velocity from pressure acceleration negates the divergence of the predicted velocity. Simplifying this results in a typical PPE for divergence [10],

$$\Delta t \nabla^2 p(t) = \rho(t) \nabla \cdot \mathbf{v}^*. \tag{2.9}$$

Grid methods can struggle to predict density, thus using equation (2.9), but since that

is not a problem in SPH, most incompressible SPH methods tend to focus on deriving equation (2.8) differently [11]. These different derivations result in the same implementation theory-wise, but terms can be rewritten where the calculations lead to improvements in performance and stability. Incompressible methods most commonly start from the density deviation PPE equation (2.8), where it is rewritten as

$$\Delta t^2 \nabla^2 p_i = \rho_0 - \rho_i^*, \tag{2.10}$$

removing the time step from the source term [10].

2.3 Predictive-Corrective Incompressible SPH

Predictive-Corrective Incompressible SPH (PCISPH) [18] uses equation (2.10), but the source term changes to $\rho_i^* - \rho_0$ to represent the local density deviation among each particle, instead of density invariance of the whole system. This is because it updates each particle until the state is solved and iterates that process until the desired density error is met, instead of solving the linear system of the state at once using a matrix. PCISPH solves for a particle's pressure through iterations by using a stiffness parameter, δ, multiplied by the current density error, $\rho_{\text{err}_i}^* = \rho_i^* - \rho_0$, resulting in an intermediate pressure value: [18]

$$\tilde{p_i} = \delta \rho_{\text{err}_i}^*. \tag{2.11}$$

The final pressure value is found by adding all the intermediate pressure values, $p_i \mathrel{+}= \tilde{p_i}$, and is iteratively solved until the desired density deviation error limit is reached. The SPH approximation of force due to pressure is denoted as [14]

$$F_i^p = -m^2 \sum_j \left(\frac{p_i}{\rho_0^2} + \frac{p_j}{\rho_0^2} \right) \nabla W_{ij}, \tag{2.12}$$

where F_i^p, m, and ∇W_{ij} denote the pressure force, mass, and gradient of the weight kernel, W, for particles i and j, respectively. PCISPH makes the assumption that each particle is applying an equal amount of pressure on each other, which simplifies equation (2.12) to $F_i^p = -m^2 \frac{2\tilde{p_i}}{\rho_0^2} \sum_j \nabla W_{ij}$ [18]. The predicted density, $\rho_i^* = \rho_i(t) + \Delta\rho_i(t)$, where the change in density is expressed as $\Delta\rho_i(t)$, requires integrating the predicted velocity to find a particle's predicted position as they use the

SPH density summation equation which is analogous to equation (3.4) [18]. This discretization is expressed as [18]

$$\rho_i^* = m \sum_j W\left(\mathbf{x}_i^* - \mathbf{x}_j^*\right). \tag{2.13}$$

Using predicted positions requires a new neighborhood update, but for performative reasons, the authors opted not to, resulting in density errors that grow over time. They use the Leap Frog integration scheme to denote this position as [18]

$$\Delta \mathbf{x}_i = \Delta t^2 \frac{\mathbf{F}_i^p}{m}. \tag{2.14}$$

The intermediate pressure variable can be derived by using the change in density. This change in density is $-\Delta t \rho(t) \nabla \cdot \mathbf{v}^*$ from the predicted density equation (2.5). Using SPH approximations with PCISPH's derivations, it is written as [18]

$$\Delta \rho_i(t) = m \left(\Delta \mathbf{x}_i(t) \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \Delta \mathbf{x}_j(t) \right). \tag{2.15}$$

Inserting the pressure force inside equation (2.14) and Δx into equation (2.15) is how intermediate pressure is solved in equation (2.11). Through that derivation, δ and β are defined as [18]

$$\delta = \frac{1}{\beta \left(\sum_j \nabla W_{ij} \cdot \sum_j \nabla W_{ij} + \sum_j (\nabla W_{ij} \cdot \nabla W_{ij}) \right)} \tag{2.16}$$

$$\beta = \Delta t^2 m^2 \frac{2}{\rho_0^2}, \tag{2.17}$$

with the caveat that there is never a particle deficiency and each particle has a filled neighborhood, which they call a "template particle." This results in accuracy errors using these "template particle" equations, but alleviates any unwanted behavior from particle deficiencies [18].

2.4 Implicit Incompressible SPH

Implicit Incompressible SPH (IISPH) [7] uses equation (2.10), where the source term enforces density invariance, keeping each particle's density constant and equal to the rest density. This is because it solves the linear system using a matrix, where pressure is not solved for one particle

at a time such as PCISPH, but rather as the entire state. Both methods follow the same logic of predicting the next time step using non-pressure forces to solve for pressure to correct the density deviation with a few key differences. To avoid costly neighborhood updates and density errors, the authors used an SPH approximation based on velocity rather than position to predict densities [7],

$$\rho_i^* = \rho_i(t) + \Delta t \sum_j m_j \mathbf{v}_{ij}^* \nabla W_{ij}, \tag{2.18}$$

where $\mathbf{v}_{ij}^*$ represents the difference in velocity between particle i and j. The pressure force is defined the same as in equation (2.12), and the equation that shows the pressure force to negate the deviation in density is denoted as [7]

$$\Delta t^2 \sum_j m_j \left(\frac{\mathbf{F}_i^p(t)}{m_i} - \frac{\mathbf{F}_j^p(t)}{m_j} \right) \nabla W_{ij} = \rho_0 - \rho^*. \tag{2.19}$$

This equation can be represented as a linear system, $\mathbf{A}(\mathrm{t})\mathbf{p}(\mathrm{t}) = \mathbf{s}(\mathrm{t})$, with one equation and one unknown pressure value for each particle, where $\mathbf{A}(\mathrm{t})$ is the matrix, $\mathbf{p}(\mathrm{t})$ denotes the pressure, and $\mathbf{s}(\mathrm{t})$ matches the source term in equation (2.19) [7]. The equation to solve for each particle in the system is denoted as [7]

$$\sum_j a_{ij} p_j = \rho_0 - \rho^*, \tag{2.20}$$

where a_{ii} is the diagonal element of the matrix which represents the displacement of the particle and its neighbors due to pressure [7]. Using a relaxed Jacobi scheme, the pressure for each particle can be found by iteratively solving the system until it meets the defined error requirement, which is expressed by [7]

$$p_i^{(l+1)} = (1-\omega)p_i^{(l)} + \frac{\omega}{a_{ii}} \left(s_i - \left(Ap^{(l)} \right)_i + a_{ii} p_i^{(l)} \right) = p_i^{(l)} + \frac{\omega}{a_{ii}} \left(s_i - \left(Ap^{(l)} \right)_i \right), \tag{2.21}$$

where l is the iteration index and ω is the relaxation factor. The term a_{ii} is denoted as [7]

$$a_{ii} = -\Delta t^2 \sum_j m_j \left(\sum_j \frac{m_j}{\rho_j^2} \nabla W_{ij} \right) \cdot \nabla W_{ij} - \Delta t^2 \sum_j m_j \left(\frac{m_i}{\rho_i^2} \nabla W_{ij} \right) \cdot \nabla W_{ij}. \tag{2.22}$$

2.4.1 Comparisons

IISPH improves on PCISPH with faster convergence and greater stability, allowing for larger time steps, but is more computationally complex as a result. Performance improvements come from more efficient matrix calculations, reduced looping overhead, and pre-computed coefficients. Fewer errors lead to faster system convergence and pressure values remaining stable. IISPH calculates the pressure for each particle rather than using a "template particle." DFSPH is heavily related to both techniques and expands upon them, combining the faster convergence and stability of IISPH and the performative computations of PCISPH. DFSPH is one of the only SPH methods that makes use of the divergence PPE denoted in equation (2.9), which uses a more common derivation, adding an additional delta time on both sides [10],

$$\Delta t^2 \nabla^2 p_i = \Delta t \rho_i \nabla \cdot \mathbf{v}_i^*. \tag{2.23}$$

Written in this form, the negative sign in the continuity equation (1.1) is implicit within the divergence of the predicted velocity.

Chapter 3

Stream Simulation in Gilligan

Stream simulation involves two main aspects—the simulation data of the fluid dynamics and creating a realistic visualization of that data.

3.1 Smoothed Particle Hydrodynamics

3.1.1 Weight Kernel

SPH uses a weight kernel to determine how much influence one particle has on another based on the distance from each other. The weight kernel can be viewed similarly to a Gaussian bell function with a normal distribution. The particle lies at the center of the peak and its influence on

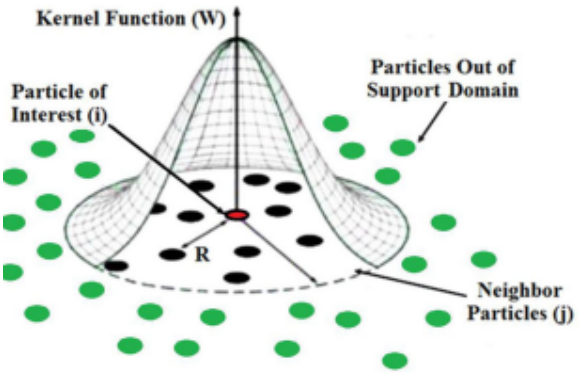

Figure 3.1: Weight Kernel Visualization [1]

both sides are equivalent because the function is symmetric, as seen in Figure 3 [1]. The maximum distance a particle can be from another and still experience influence is called the smoothing length, often denoted by h [10]. The weight kernel is also bound by its own support radius, where distances beyond the kernel radius values are set to zero. Weight kernels are one of the biggest factors of stability and accuracy in a simulation, which directly affects the flow of a fluid [1]. A popular weight kernel in SPH uses a cubic spline [10],

$$W(\mathbf{r}, h) = \frac{8}{\pi h^3} \begin{cases} 6(q^3 - q^2) + 1, & 0 \leq q \leq \frac{1}{2}, \\ 2(1-q)^3, & \frac{1}{2} < q \leq 1, \\ 0, & \text{otherwise}, \end{cases} \tag{3.1}$$

where $\mathbf{r}$ is the distance between two particles $(\mathbf{x}_i - \mathbf{x}_j)$, $\frac{8}{\pi h^3}$ is the normalization factor for three dimensions, h is the kernel radius, and q is $\frac{\|\mathbf{r}\|}{h}$. This weight kernel is often used because its smoothing length is equivalent to the kernel support radius, which is why they share the h notation. In this implementation, the kernel radius is set to 0.1 m and is commonly shortened to W_{ij}. The kernel function only returns a scalar to determine the weight of the influence, so in order to retain spatial information, the gradient of the kernel is computed to determine the influence in a direction. To avoid potential divisions by zero and floating point errors when the distance between particles is marginal, a check is performed before the gradient calculation is done comparing $\|\mathbf{r}\|$ to an ϵ, set to 10^{-9}. If the magnitude of $\mathbf{r}$ is less than ϵ, the gradient will return a zero vector, and if it is equal to or larger than ϵ, the gradient is computed as

$$\nabla W(\mathbf{r}, h) = \frac{48}{\pi h^4} \begin{cases} q\,(3q-2)\,\hat{\mathbf{r}}, & 0 \leq q \leq \frac{1}{2}, \\ -(1-q)^2 \hat{\mathbf{r}}, & \frac{1}{2} < q \leq 1, \\ (0,0,0), & \text{otherwise}. \end{cases} \tag{3.2}$$

3.1.2 SPH Approximations

SPH approximates spatial field quantities and spatial differential operators by summing over all particles at a given position and is denoted by [10]

$$(A * W)(\mathbf{x}_i) \approx \sum_{j \in \mathcal{F}} A_j \frac{m_j}{\rho_j} W_{ij}, \tag{3.3}$$

where A is the quantity or operator, W is the weight kernel, $\mathbf{x}_i$ is the current position, m is mass, ρ is density, and $\mathcal{F}$ is the set of all particles. Typically, a particle's radius is set to $\frac{1}{4}$ of the kernel radius, making the volume of a particle in one dimension its diameter [10]. Mass is assigned to all particles at the start of the simulation, and for this implementation, it was set to $(volume)^3$ * 0.8, where the volume is slightly reduced to avoid high pressure levels at the start of the simulation [2].

3.1.3 Neighborhood Search

Looping through all the particles in a simulation has a runtime complexity of O(n^2), but given the nature of SPH, only particles within the kernel's radius will contribute to one another. The particles inside of that distance are labeled as neighbors, and all other particles are unneeded for the calculation. When only using a particle's neighbors, the runtime complexity is brought down to O(mn), where m is the number of particles in a neighborhood. To find these neighbors, a neighborhood search algorithm is used, with one of the most common making use of an occupancy grid or uniform grid that can be built in O(n) and queried in O(1) [6]. A 3D grid forms a cube-like structure, seen in Figure 3.2, defined by a lower left hand corner, upper right hand corner, and a cell size. The size of the cells is the support kernel diameter. Therefore, any particle within one cell only has to check the adjacent cells for its neighborhood. The downside of the uniform grid is that populating it in parallel is not thread safe, cache hits for retrieving the neighborhood can be low, and, in large simulation environments, there can be a large amount of memory wasted on empty cells [6]. The uniform grid was chosen in this simulation as the environment only encompasses the streambed, and the benefits of other methods, such as those that are tree-based, introduce more overhead building and querying diminishing that are only worth it for larger environments [6].

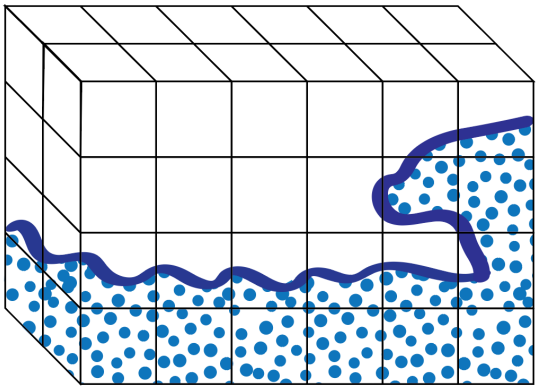

Figure 3.2: Occupany Grid

3.1.4 Density

Using the SPH approximation equation, the density of a particle can be calculated by replacing A with ρ, creating the following equation [10],

$$\rho_i = \sum_j \frac{m_j}{\rho_j} \rho_j W_{ij} = \sum_j m_j W_{ij}, \tag{3.4}$$

where j is now referencing the neighborhood of particle i which includes itself. In water SPH simulations, the goal is to maintain each particle at its rest density, typically denoted as ρ_0, which is 1000 kg m^{-3} when water is not moving [10].

3.1.5 Viscosity

From the Navier-Stokes equation (1.3), the dynamic viscosity is found through multiplying the kinematic viscosity coefficient, ν, by the Laplacian of the velocity. Monogan shows the Laplacian of the velocity can be approximated as

$$\nabla^2 \mathbf{v}_i = 2(d+2) \sum_j \frac{m_j}{\rho_j} \frac{\mathbf{v}_{ij} \cdot \mathbf{x}_{ij}}{||\mathbf{x}_{ij}||^2 + 0.01h^2} \nabla W_{ij}, \tag{3.5}$$

where d represents the dimensions and h is the kernel radius [15]. In this implementation, the kinematic viscosity coefficient is set to 0.01 $m^2 s^{-1}$, which is common for low viscous fluids like water [10]. This is a more accurate representation of real viscosity that is discretized through finite

differences compared to artificial viscosity, and it is able to conserve linear and angular momentum while producing more realistic flows [10].

3.1.6 Courant–Friedrichs–Lewy Condition

The original solver in Gilligan took user input to set a fixed time step. This can destabilize an SPH simulation as certain particles may start to accelerate too quickly in larger time steps, introducing more and more errors. To offset this, the Courant–Friedrichs–Lewy (CFL) condition is a dynamic time step based on the fastest particle in the simulation, where if it is moving too fast for the preferred time step given by the user, it will lower the time step to ensure the particle does not move more than 40% of its diameter. The CFL condition is denoted as [2]

$$\Delta t \leq \frac{0.4 \cdot d}{|\mathbf{V}_{\max}|}, \tag{3.6}$$

where 0.4 is a tested constant to achieve stable results [14], d is the diameter of the particle and Δt is bounded by 0.0001 s and 0.005 seconds.

3.1.7 Boundary Handling

With it being customary for geometry to be composed of triangles, boundary handling can consist of testing to see if any particles hit a triangle. One of the simplest forms of collision detection is determining where a particle is in respect to an infinite plane, $f(\mathbf{x}) = \hat{\mathbf{n}}_P \cdot (\mathbf{x} - \mathbf{x}_P)$, where $\hat{\mathbf{n}}_P$ and $\mathbf{x}_P$ is the normal of the plane and a point on the plane, respectively, and $\mathbf{x}$ is the position of the particle. When the function equals zero, the particle is directly on the plane, and greater than or less than zero is above or below the plane, respectively. The position of a particle before and after integration can be compared to see if they are on opposite sides of the plane, or if the updated position has hit the plane directly.

The particle cannot be reflected off the plane based solely on this test, as being integrated by a discrete time step allows for a particle to hit the plane before the full time of the time step has elapsed. The time of the hit needs to be calculated and the rest of the time step can be used to reflect the particle. The time of the collision can be derived from the position where the collision took place. The hit position is defined as $\mathbf{x}_H = \mathbf{x}_S + \mathbf{v}_S \frac{\hat{\mathbf{n}}_P \cdot (\mathbf{x}_P - \mathbf{x}_S)}{\hat{\mathbf{n}}_P \cdot \mathbf{v}_S}$ [19], where $\mathbf{x}_S$ and $\mathbf{v}_S$ is the position and velocity of the particle before integration. To integrate the start position

to the hit position, it is simply $\mathbf{x}_H = \mathbf{x}_S + \mathbf{v}_S \Delta t_H$. Solving for the time of the hit is denoted by $\Delta t_H = \frac{\mathbf{v}_S \cdot (\mathbf{x}_H - \mathbf{x}_S)}{\|\mathbf{v}_S\|^2} = \frac{\hat{\mathbf{n}}_P \cdot (\mathbf{x}_P - \mathbf{x}_S)}{\hat{\mathbf{n}}_P \cdot \mathbf{v}_S}$ [19]. Once a particle hits the plane, its velocity will change due to inelastic reflection, $\mathbf{v}_R = C_s \mathbf{v}_S - (C_s + C_r)\hat{\mathbf{n}}_P(\hat{\mathbf{n}}_P \cdot \mathbf{v}_S)$ [19], where C_s and C_r are coefficients bounded by 0 and 1 for stickiness and restitution, respectively. The position at the end of this reflection is $\mathbf{x}_R = \mathbf{x}_H + \mathbf{v}_R(\Delta t - \Delta t_H)$. Within a given time step, a particle may hit multiple planes, so all the Δt_H will be calculated and the smallest time hit determines which plane was hit first. The particle will be reflected off that plane and tested for any other collisions following the same routine until the particle has traveled the entire time step.

A point is on the infinite plane of a triangle if it satisfies $\hat{\mathbf{n}}_T \cdot (\mathbf{P} - \mathbf{P}_0) = 0$, where $\hat{\mathbf{n}}_T$ is the normal of the triangle, $\mathbf{P}$ is the point being tested, and $\mathbf{P}_0$ is a vertex of the triangle. Points on an infinite plane of a triangle can be written as an explicit parametric equation, $\mathbf{P} = w\mathbf{P}_0 + u\mathbf{P}_1 + v\mathbf{P}_2$ [19], where $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ are the vertices of the triangle and the weights w, u, and v sum to one. The point of the plane can be rewritten as $\mathbf{P}(u, v) = \mathbf{P}_0 + u\mathbf{e}_1 + v\mathbf{e}_2$, where $\mathbf{e}_1$ and $\mathbf{e}_2$ are two edges of the triangle [19]. The area of the triangle can be found through Barycentric Coordinates, which constrain the coefficients u and v in the point parametric question such that $0 \leq u \leq 1$, $0 \leq v \leq 1$, and $0 \leq u + v \leq 1$. Given a particle position, it can be tested for collisions on the infinite planes of triangles. If a particle hits one of those planes, it can be tested if its inside the triangle through Barycentric Coordinates. Those coordinates can be found through $u = \frac{\hat{\mathbf{n}}_T \cdot ((\mathbf{P} - \mathbf{P}_0) \times \mathbf{e}_2)}{\|\hat{\mathbf{n}}\|^2}$ and $v = \frac{\hat{\mathbf{n}}_T \cdot (\mathbf{e}_1 \times (\mathbf{P} - \mathbf{P}_0))}{\|\hat{\mathbf{n}}\|^2}$ [19], proving the particle is inside the triangle if they are correctly constrained. Collision detection is the same as the infinite plane logic, with the added Barycentric Coordinates test.

3.1.7.1 BVH

Comparing a particle to every triangle is unnecessary, especially for large geometries where particles can be nowhere near colliding with certain areas, and can be optimized by using a Bounding Volume Hierarchy (BVH) to increase performance and reduce calculations. A BVH is a tree structure that makes use of Axis Aligned Bounding Boxes (AABB) as leaf nodes. An AABB is a region of space designated by a lower left hand corner and upper right hand corner. The entire tree structure is one AABB that encompasses all geometry and the leaf nodes split up this AABB into smaller ones. The concept is to divide the original AABB across one axis evenly and create AABBs for the smaller regions that contain triangles [13]. An AABB is created for each triangle and an AABB

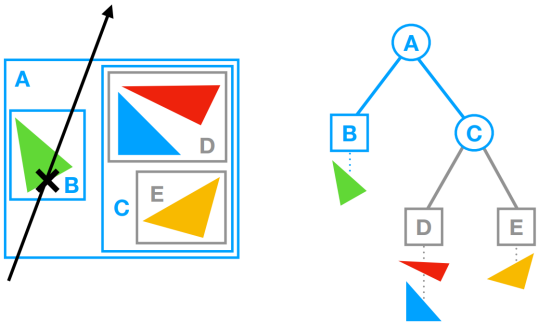

Figure 3.3: An example of a BVH structure. The left-hand side showcases geometry being inside of AABBs with a ray intersecting them. The right-hand side showcases the tree structure of all the AABBs queried and the geometry returned from the ray intersection is only the triangle in box B [13]

intersection test can be performed to see if a new AABB must be created [21]. The process continues with a new division across a different axis until a user specified number of divisions have occurred. Each AABB contains a list of triangles for its region, so the only triangles needed to be tested for a particle are the ones in the AABB where the particle resides. The particle position is tested from the root node to the lowest level AABB, where its position is contained within the lower left and upper right corner. However, since the triangle collision detection uses the real delta time in which a particle is hit, a ray must be created from the start position and end position of a particle [13]. All the AABBs this ray intersects is used for accurate collision detection.

3.2 Divergence-Free SPH

DFSPH uses two pressure solvers, one to keep constant density and another to enforce a divergence-free velocity field [2]. This differs from the other SPH methods that only use a constant density solver. Using two solvers makes it more computationally expensive, but solving for one helps to lower the number of iterations needed to meet the density error criteria in the other. DFSPH is also more performative than PCISPH and IISPH because the iterations per solver can be kept low and variables can be reused for both solvers [2]. The second solver improves accuracy by enforcing a divergence-free velocity field, enhancing stability, and allowing larger time steps. With larger time steps, the costly neighborhood search in SPH can be performed less frequently. An entire overview of a DFSPH simulation can be seen in Listing 3.1 [2].

Listing 3.1: DFSPH Simulation Algorithm

```
void DFSPH_Solver()
    while (t < t_max) do // Start sim loop
        // Populate Occupancy Grid
        for all particles i do
            find neighborhoods N_i
        for all particles i do
            compute densities ρ_i
            compute factors α_i
        correctDivergenceError(α, v*) // Fulfill Dρ/Dt = 0
        // Compute Non-pressure accelerations
        for all particles i do
            compute non-pressure accel a(t)
        adapt time step Δt to CFL condition
        for all particles i do // Predict v_i*
            v_i* = v_i + aΔt
        correctDensityError(α, v_i*) // Fulfill ρ* − ρ_0 = 0
        for all particles i do // Update pos
            x_i(t + Δt) = x_i(t) + Δt v_i*
        handle_collisions()
```

This algorithm loops for a set amount of time, where the neighborhoods are calculated first, which is needed to compute the densities and the factor terms that are solely reliant on positions. For the first run, given any initial starting velocities, the velocity field is solved to be divergence-free. Following the logic of solving a PPE, the particles' acceleration from non-pressure forces is calculated and the predicted velocities are integrated based on the time step of the CFL condition. From there, the constant density solver fixes the deviation in density, which updates the predicted velocity with the acceleration from pressure. The particles can then be integrated for the updated positions and boundary handling can occur, which allows the loop to repeat until the simulation is over. The order of the solvers does not matter and the DFSPH method does not require the constant density solver from their paper, as any constant density solver can be used, such as the one from IISPH [3].

The SPH approximation for equation (2.4) is implemented, differentiating it with respect to position and allowing the pressure gradient to be discretized as [2]

$$\nabla p_i = \kappa_i^v \nabla \rho_i = \kappa_i^v \sum_j m_j \nabla W_{ij}, \tag{3.7}$$

where κ_i^v is the stiffness parameter that needs to be solved for, instead of user input such as in an EOS solver. Replacing the pressure gradient inside the Navier-Stokes equation (1.3) with the SPH

approximation, acceleration from pressure is written as [2]

$$\mathbf{a}_i^p = -\frac{1}{\rho_i}\kappa_i^v \sum_j m_j \nabla W_{ij}. \tag{3.8}$$

Particle i's pressure force acts upon its neighbors and, to conserve momentum, the forces must be symmetric and fulfill equation $F_i^p + \sum_j F_{j\leftarrow i}^p = 0$ [2]. This means that particle i's neighbors should exert the same pressure force onto particle i ($F_{i\leftarrow j}^p$) in the opposite direction that particle i exerts on them. The pressure acceleration of particle i on its neighbors is denoted as [2]

$$\mathbf{a}_{j\leftarrow i}^p = -\frac{1}{\rho_i}\frac{\partial p_i}{\partial \mathbf{x}_j} = \frac{1}{\rho_i}\kappa_i^v m_j \nabla W_{ij}. \tag{3.9}$$

Following the improvements of IISPH, the density derivative is calculated using velocities over position [3],

$$\frac{D\rho_i}{Dt} = \sum_j m_j(\mathbf{v_i} - \mathbf{v_j})\nabla W_{ij}. \tag{3.10}$$

Replacing the velocity terms in equation (3.10) with the pressure acceleration for particle i in equation (3.8) and the pressure acceleration that acts upon particle's i neighbors in equation (3.9) allows κ_i^v to be solved for as [2]

$$\kappa_i^v = \frac{1}{\Delta t}\frac{D\rho_i}{Dt}\alpha_i. \tag{3.11}$$

In solving for κ_i^v, the coefficient α_i is defined as [2]

$$\alpha_i = \frac{\rho_i}{\left|\sum_j m_j \nabla W_{ij}\right|^2 + \sum_j \left|m_j \nabla W_{ij}\right|^2}, \tag{3.12}$$

where it solely relies on particle positions and is referred to as the factor. This derivation can suffer from instabilities due to particle deficiencies, but clamping α_i to 10^{-5} resolves such issues without being noticeable [2]. The total acceleration due to pressure for particle i is also based on the pressure forces its neighbors act upon itself, which is computed analogously to equation (3.9), since the pressure forces are symmetric [2],

$$\mathbf{a}_{\mathbf{i,total}}^{\mathbf{p}} = \mathbf{a}_i^p + \sum_j \mathbf{a}_{i\leftarrow j}^p = -\sum_j m_j\left(\frac{\kappa_i^v}{\rho_i} + \frac{\kappa_j^v}{\rho_i}\right)\nabla W_{ij}. \tag{3.13}$$

Note that this resembles equation (2.12) previously defined to locate pressure in SPH. Since α_i has ρ_i in the numerator and the κ_i^v have denominators with ρ_i, the ρ_i coefficient can be dropped from both, making factor have a numerator of one and leaving a simplified acceleration pressure equation,

$$\mathbf{a}^{\mathbf{p}}_{\mathbf{i,total}} = -\sum_j m_j(\kappa_i^v + \kappa_j^v)\nabla W_{ij}. \tag{3.14}$$

When calculating the pressure for a particle, if $\kappa_i^v + \kappa_j^v$ is less than 10^-5, the computation is ignored to increase numerical stability and performance, and nothing is lost since the contribution is insignificant.

3.2.1 Divergence-Free Velocity Field

To solve for a divergence-free velocity field, the density derivative, $\frac{D\rho}{Dt}$, should equal zero. The contribution based on pressure forces to correct the divergence source term in equation (2.23) is determined by [2]

$$\frac{D\rho}{Dt} = -\Delta t \sum_j m_j(\mathbf{a}_i^p - \mathbf{a}_{j\leftarrow i}^p)\nabla W_{ij}. \tag{3.15}$$

Listing 3.2: Divergence Solver

```
void correctDivergenceError(α_i, v*_i)
    for all particles i do
        compute Dρ/Dt // density derivative (source term, s_i)
        compute κ^v_i
    while(ρ_avgError ≤ η^v)
        for all particles i do
            compute pressure_acc()
        for all particles i do
            compute corrective_pressure_force()
            density_error = s_i + corrective_pressure_force
            κ^v_i += 0.5 * density_error * 1/Δt α_i
            ρ_error += max(density_error,0)
        ρ_avgError = ρ_error / num_particles
    for all particles i do
        compute pressure_acc()
        v*_i = v*_i + a_pressure Δt
```

Listing 3.2 showcases the algorithm for the divergence solver [2], where η^v is a user defined threshold that dictates how accurate the density deviation can be and is often set to 0.1% of the rest

density times $\frac{1}{\Delta t}$. The idea being that the average density error should be less than the proportion of the rest density based on the simulation's time step. The density error comes from placing the corrective pressure forces on the left-hand side in equation (3.15), which is adding the source term with the corrective pressure forces. This means the change in density due to pressure advection negates the change in density due to non-pressure advection. The density error used for the while loop test is clamped to zero and automatically set to such if the number of neighbors is less than 20 [2]. The average is found by summing every particle's density error and dividing by the number of particles. The initial calculation of κ_i^v uses the density derivative clamped to zero and the density derivative is set to zero if the number of neighbors for a particle is less than 20 [2]. This is to start the while loop iterations solving solely for compression regardless of particle deficiencies. The final result for κ_i^v is solved in a Jacobi-like fashion and given the nature of approximations, numerical errors will occur, and a minimum and maximum iteration limit is set to two and 100, respectively. The density error is reduced by gradually increasing κ_i^v scaled by half of the density error, which will increase pressure to resolve the compression. During each iteration, a particle's pressure acceleration is calculated at the start, allowing for an independent update of κ_i^v to be completed in parallel [2]. The current density error for each particle during the iterations is not clamped and is halved in the calculation for κ_i^v to even out the pressure of a particle's neighborhood and create a smoother convergence without overshooting [2]. The κ_i^v term is clamped to zero which avoids negative pressures that can act unrealistically in SPH [10] and helps areas with particle deficiencies, such as the surface. When κ_i^v reaches the density error criteria, it is used to recalculate the particle's acceleration due to pressure and the predicted velocity is integrated with this pressure acceleration. Since α_i is solely reliant on the current position of the particle, it can be pre-computed before the iterations start and reused for the next solver [2].

3.2.2 Constant Density

To solve for constant density, this solver uses the source term from PCISPH, $\rho_i^* - \rho_0$, and the predicted density is solved in the same way as IISPH [2],

$$\rho_i^* = \rho_i + \Delta t \frac{D\rho_i}{Dt}. \tag{3.16}$$

Replacing the material density derivative in equation (3.10) with $\rho_i^* - \rho_0$ results in a new equation that defines the scalar contribution based on pressure forces to correct the new source term [2],

$$\rho_i^* - \rho_0 = -\Delta t^2 \sum_j m_j(\mathbf{a}_i^p - \mathbf{a}_{j \leftarrow i}^p)\nabla W_{ij}. \tag{3.17}$$

Deriving the corrective pressure force without relying on integrated positions avoids having to assume the neighborhood is filled, such as with the "template particle" in PCISPH. This is the same as equation (2.19), except the source term is flipped, resulting in the negative on the right-hand side. Moving all terms to one side should result in zero, and when it is discretized, it will hold as the density error remaining after corrective forces have been applied. Solving for the stiffness parameter in the same fashion as equation (3.10) results in [2]

$$\kappa_i = \frac{1}{\Delta t^2}(\rho_i^* - \rho_0)\alpha_i. \tag{3.18}$$

To have a distinction between the constant density and divergence-free solver, the stiffness parameter for the constant density solver will be denoted solely as κ_i.

Listing 3.3: Constant Density Solver

```
1  void correctDensityError(α_i, v_i*)
2      for all particles i do
3          compute ρ_i* //predict densities (source term, s_i)
4          compute κ_i
5      while(ρ_avgError ≤ η)
6          for all particles i do
7              compute pressure_acc()
8          for all particles i do
9              compute corrective_pressure_force()
10             density_error = s_i + corrective_pressure_force
11             κ_i += 0.5 * density_error * 1/Δt² α_i
12             ρ_error += max(density_error,0)
13         ρ_avgError = ρ_error / num_particles
14     for all particles i do
15         compute pressure_acc()
16         v_i* = v_i* + a_pressure Δt
```

Listing 3.3 showcases the algorithm for the constant density solver [2], where η is a user defined threshold dictating how accurate the density deviation can be and is often set to 0.01% of the rest density. This solver is set up almost exactly as Listing 3.2 and uses a minimum and maximum iteration limit set to two and 100. The only difference is the equation change with regard to the

new source term and not setting the predicted densities to zero in a particle deficiency. The change in source term means κ_i is being increased so that the density deviation from pressure advection negates the density deviation from non-pressure advection.

3.3 Marching Tetrahedra

One of the first techniques for creating a triangle mesh from volumetric data is called Marching Cubes (MC) and was introduced in 1987 [12]. Marching Tetrahedra (MT) [4] derives from MC and eliminates some of the ambiguities that arise when creating a surface from a cube representation. The use of MT in Gilligan is implemented on a 3D rectangular sparse grid, where scalar field values are sampled and stored on the grid points in a process called stamping [19]. A sparse grid partitions a full grid, similar to the occupancy volume grid, into blocks which are only allocated memory if they are used to lower memory costs. The scalar field can be derived from a Signed Distance Function (SDF), which represents the distance of any point from the surface of the implicit function. Gilligan uses the convention of positive values being inside and negative values being outside. A grid stamped with an SDF is referred to as a level set, where positions in the grid, but not on a grid point, are approximated through trilinear interpolation. Particles are represented by spheres with the SDF defined as $r - |\mathbf{x}|$, where r is the sphere radius and $\mathbf{x}$ is the position in space [19]. MC generates a polygonal representation of an iso-surface of the level set from a defined iso-value. To generate a surface from a level set, the iso-value should be zero. MT follows MC by starting from a cube representation of the 3D grid cells, which can be referred to as voxels, made from the eight grid points. In MC, the iso-surface can intersect a voxel in 256 distinct ways which can be reduced down to 14 distinct patterns, but it can also create holes in the mesh due to face ambiguities [12]. In MT, a voxel is split into six tetrahedrons, where one tetrahedra has 16 distinct intersections that can be reduced down to eight distinct patterns and removes the face ambiguities that arise when using squares [4]. MT can suffer from ambiguities on shared edges, which can be reduced by subdividing a voxel into more tetrahedrons. Gilligan uses the typical 12 tetrahedra subdivision for smoother interpolations and a more robust mesh [19]. If none of the grid points equal the iso-value, an edge containing points above and below will contain the iso-value and the surface intersection point can be found through linear interpolation. This point is represented as, $\text{intersection\_v} = v_1 + \frac{(\text{isovalue} - s_1)(v_2 - v_1)}{s_2 - s_1}$, where v represents the grid points' position and s represents

the grid points' value [12]. Based on the edges where an intersection point occurs, a look-up table can be used to form triangles based on the eight patterns [4]. Once triangles form, each vertex's normals can be computed by the gradient of the SDF. The gradient of the triangle vertex can be approximated by finding the gradient of each grid point through central differences [19],

$$\frac{\partial f(\mathbf{x})}{\partial x} \approx \frac{f(x+\Delta x, y, z) - f(x-\Delta x, y, z)}{2\Delta x} \tag{3.19}$$

$$\frac{\partial f(\mathbf{x})}{\partial y} \approx \frac{f(x, y+\Delta y, z) - f(x, y-\Delta y, z)}{2\Delta y} \tag{3.20}$$

$$\frac{\partial f(\mathbf{x})}{\partial z} \approx \frac{f(x, y, z+\Delta z) - f(x, y, z-\Delta z)}{2\Delta z}, \tag{3.21}$$

where f represents the SDF at a spatial position and Δ x, y, z are the grid cell sizes for each dimension. The gradient of the triangle vertex can be found by interpolating along the edge and normalizing the vector.

3.4 Global Illumination

Global Illumination (GI) [17] is the physically-based rendering technique that mimics how radiance is transported in the real world. The light reflected back to a human eye or camera can be represented in the rendering equation established in 1986. The rendering equation is expressed as [9]

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{S^2} f(\mathbf{p}, \omega_o, \omega_i) L_i(\mathbf{p}, \omega_i) |\cos\theta_i| \, d\omega_i, \tag{3.22}$$

where L_o represents the outgoing radiance from a particular point in space, $\mathbf{p}$, based on the angle to the camera, ω_o. The total radiance contributing to L_o is made up of L_e, the emitted radiance from the point itself and the integral of incoming radiance, L_i, from a point and angle, ω_i, scaled by the Bidirectional Scattering Distribution Function (BSDF), f, and geometry term, $|\cos\theta_i|$, in all directions on a sphere, S^2, taken over the differential solid angle, $\mathrm{d}\omega_i$. The geometry term expresses how much of the incoming radiance is hitting the surface, and the BSDF expresses how much of the radiance that is reflected or transmitted ends up back at the camera to account for materials that are translucent [17]. This integral makes up the direct illumination, radiance from the light

source, indirect illumination, and radiance reflected from other surfaces. The rendering equation cannot be solved analytically in the majority of cases, leading to the use of the numerical integration technique, Monte Carlo Integration, that approximates the integral through random sampling [17]. One method of utilizing this is called path tracing, where a given pixel has a number of randomly positioned rays based on the sampling rate cast from the camera. These rays form paths bouncing off surfaces and accumulating the radiance contributions until it reaches the light source or the bounce limit. At each surface hit, the rendering equation is approximated and a new ray continues in a random direction based on the BSDF. The averaging of the accumulation of radiance for all the sampled paths determines the color of the pixel.

3.4.1 Shaders

Different surface materials interact with light differently. To determine the random direction for a new ray, how much radiance is reflected, refracted, or absorbed for the ray that intersected a surface requires the BSDF to be material-specific. For an offline renderer, the defining of the BSDF for each material is referred to as shaders. Gilligan uses shaders in a stack data structure, where a given material goes through the stack to define a BSDF in pieces. The water material consists of two shaders, one to attenuate and absorb light for the transparency inside the surface and another to reflect light on the surface and show glints.

3.4.1.1 Water Volume Shader

The water volume shader checks for rays that hit the surface and continue inside to correctly model attenuation. The attenuation is based on Beer's Law which is defined as [19]

$$T = e^{-\sigma d}, \tag{3.23}$$

where T is transmittance (radiance that is not absorbed), σ is absorption coefficient for the material, and d is the distance traveled inside the surface. Based on Beer's Law, the color accumulated for the ray is blended, accounting for the fact that light loses intensity as it travels through water.

3.4.1.2 Ocean Surface Shader

The ocean surface shader checks if the ray comes from below or above the surface to calculate the reflected ray. If refraction is possible, then reflection is based on Fresnel Reflectance Equations [19]

$$R_s = \left(\frac{\eta \cos\theta_i - \cos\theta_t}{\eta \cos\theta_i + \cos\theta_t} \right)^2 \tag{3.24}$$

$$R_p = \left(\frac{\eta \cos\theta_t - \cos\theta_i}{\eta \cos\theta_t + \cos\theta_i} \right)^2 \tag{3.25}$$

$$R = \frac{R_s + R_p}{2}, \tag{3.26}$$

where the total reflectance, R, is the combination of the reflectance of the s-polarized light, R_s, and the p-polarized light, R_p. Equation (3.24) and (3.25) is defined where η is the refractive index for the medium and θ_i and θ_t are the angles of the incidence and transmitted based on the direction of the ray (above or below the surface). Refraction can be checked through Snell's Law that states [19]

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t. \tag{3.27}$$

The ratio of the refractive indices for the incidence and transmitted (η_i, η_t) is equal to the ratio of the sines of the angle of incidence and transmitted. If the computed value of $\sin\theta_t$ exceeds 1, refraction is not possible [19]. Based on equation (3.27), the transmitted direction can be found and combined with the Fresnel equations to create a transmitted ray that accounts for both reflection and refraction to travel through the medium.

3.4.1.3 Cull by Neighborhood

The Cull by Neighborhood shader is used to remove a particle's visibility during rendering. When a ray hits a surface, the occupancy grid cell for that hit position is located to get the number of particles in the cell. The neighborhood for that cell, which are the adjacent cells, have their particle counts summed together with the cell hit by a ray to see the number of particles in the neighborhood of the ray hit. If the neighborhood is below the user-defined threshold, the ray is not shaded and a new ray continues in the same direction. This gives the user the ability to assign a threshold based on the neighborhood count and determine how densely packed particles need to be for rendering. This culls particles that are isolated without removing splashes, but leaves the issue

of struggling to cull the unwanted particles near a large sum of neighbors without interfering with non-flyaway particles.

Chapter 4

Results

4.1 Particle Simulation

The simulation is run using a triangle geometry model of the Hunnicutt Creek with the intent to fill the streambed as seen in Figure 4.1. Five sources randomly emit 500 particles in a sphere shape every frame until frame 1900 seen in Figure 4.2, resulting in almost four million particles. The geometry of the streambed allows particles to leave the domain (outside of the occupancy grid), and they are culled from the simulation. To account for this loss, the furthest source in the back keeps the emission rate of 500 plus the number of culled particles. After frame 1900, four sources turn off, leaving only the source furthest in the back (orange emitter) on and creating a steadily increasing flow down the streambed. To improve performance, the first 1900 frames have a fixed time step of 0.01 seconds to fill the streambed in less frames and is set back to the CFL controlled time step with a minimum of 0.005 seconds. To generate each simulated frame, a Python script is run to create a .pba file that writes each particle's ID, position, velocity, and color. In order to save memory, the file is compressed, averaging around 250 MB per frame. The next frame that needs to be simulated reads the last .pba file and solves for the new state, allowing the simulation to be queried on a job scheduler via a given frame set. Jobs were set to use 80 CPU cores and required 20 GB of memory using OpenMP for parallelization. Both solvers are set to 100 iterations with the density error threshold for the density change rate error being 0.1% for the divergence solver and the density error threshold for volume compression being 0.01% for the constant density solver. To visualize the .pba frames quickly for testing, particles are rendered as tetrahedron using

Figure 4.1: 3.5 million particles rendered as grey tetrahedron using Gilligan's lambertian shader.

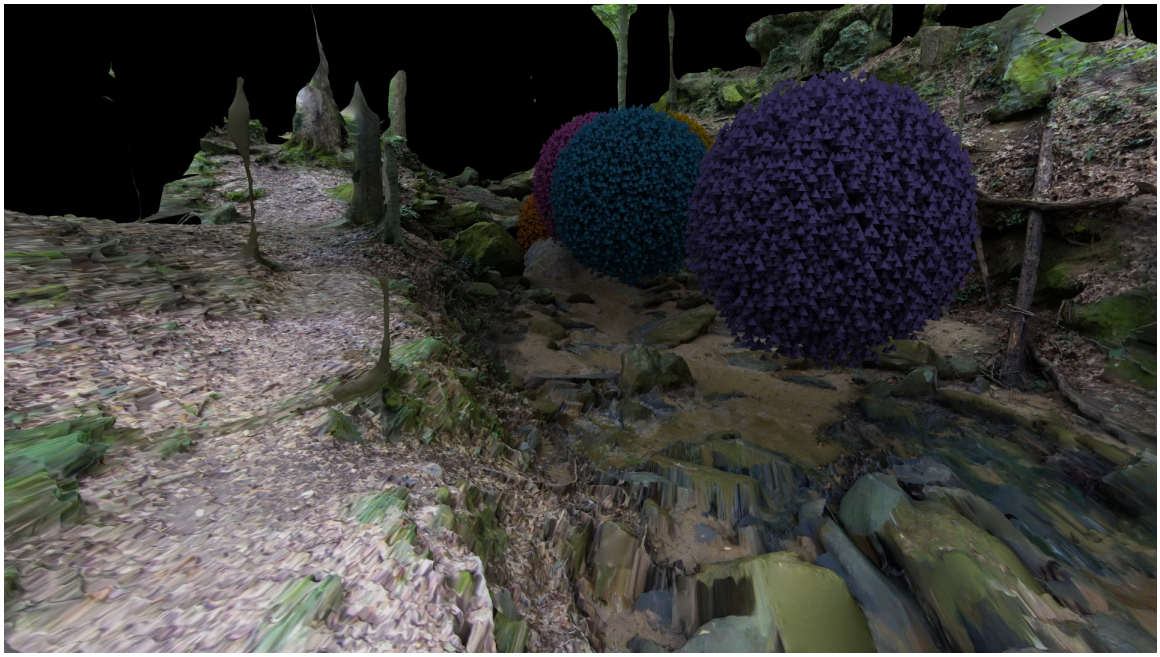

Figure 4.2: Five source emitters initializing particle colors to orange, yellow, pink, blue, and purple.

Figure 4.3: 3.5 million particles rendered using Gilligan's lambertian shader accounting for source emitter color.

a spectral lambertian shader from Gilligan based on their color for visual clarity and without the use of the skybox modeled from the environment [19]. Using one color limited the visual clarity and small details of the fluid flow were lost, which led to coloring the particles based on their respective emitters. Velocities were left unclamped to test for stability, and the flyaway particles seen in Figure 4.3 could be lessened by restricting speeds that may come from pressure and bouncing off geometry.

An overview of the simulation data can be seen in Table 4.1, referencing each frame with the density error per solver and the amount of iterations it took to solve for each solver in correlation with the number of particles. Divergence error (Div. Error) and divergence iterations (Div. Iter.) correspond with the divergence-free solver where the error is in units of $kg/(m^3 \cdot s)$. Density error and density iterations (Density Iter.) correspond with the constant density solver where the error is in units of kg/m^3. Solve Time represents the total time it took to solve for both solvers in seconds.

The average density change rate error ($kg/(m^3 \cdot s)$) should be kept below 0.1% * rest density (1000 kg/m^3) * $1/\Delta t$. When the time step is 0.01 and 0.005 seconds, the values are 100 and 200 $kg/(m^3 \cdot s)$, respectively. Since smaller time steps result in more updates, a stricter threshold is not required. Figure 4.4 and Table 4.2 showcases the solver's reliability in keeping the density

Frame	Div. Error	Div. Iter.	Density Error	Density Iter.	Solve Time (s)	Particles
1	0.000	1	0.000	2	0.759	2493
278	87.452	5	0.091	11	29.873	650907
555	94.859	7	0.100	26	98.588	1282515
832	90.239	9	0.094	41	268.639	1905617
1109	96.911	10	0.098	75	723.545	2506157
1386	87.272	11	0.097	67	915.958	3098433
1663	94.572	10	0.099	73	1356.452	3509475
1940	108.545	9	0.099	67	1566.711	3674176
2217	122.518	7	0.099	61	1776.971	3838876
2494	136.490	5	0.099	55	1987.230	4003577
2771	150.463	3	0.100	49	2197.490	4168278
3048	161.406	2	0.099	44	2223.430	4327870
3325	163.929	2	0.100	44	1815.330	4463594

Table 4.1: Simulation Results

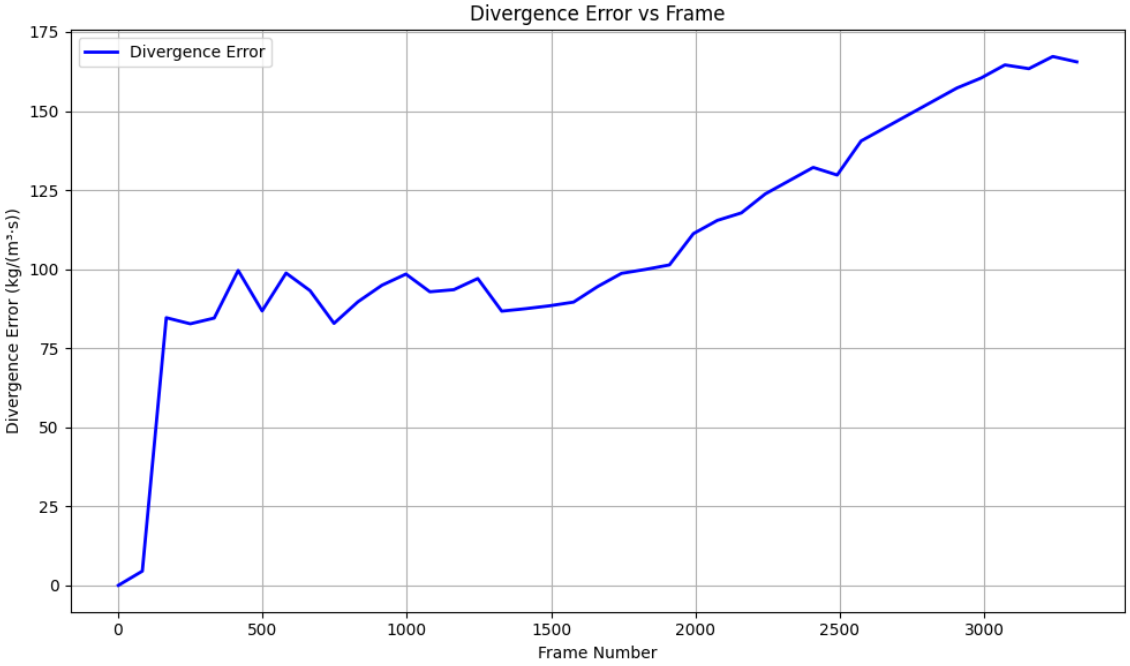

Figure 4.4: Divergence Error Graph

Divergence Error	Value ($\mathrm{kg}/(\mathrm{m}^3 \cdot \mathrm{s})$)
Max Div. Error	199.8810
Min Div. Error	0.0000
Mean Div. Error	109.6179
Std Dev. Div. Error	34.6576

Table 4.2: Divergence Solver Error Statistics

Divergence Iterations	Value
Max Div. Iterations	11.0000
Min Div. Iterations	1.0000
Mean Div. Iterations	6.6169
Std Dev. Div. Iterations	3.1273

Table 4.3: Div. Solver Iterations Statistics

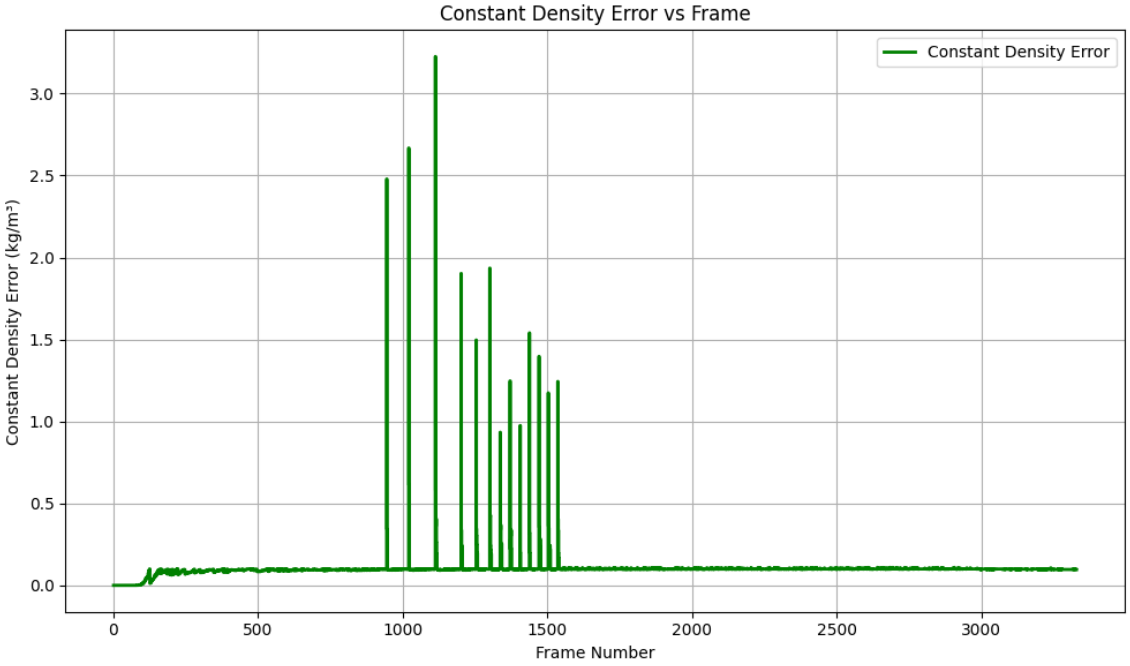

Figure 4.5: Constant Density Error Graph

Density Error	Value (kg/m^3)
Max Density Error	3.2264
Min Density Error	0.0000
Mean Density Error	0.1021
Std Dev. Density Error	0.1137

Table 4.4: Density Solver Error Statistics

Density Iterations	Value
Max Density Iterations	100.0000
Min Density Iterations	2.0000
Mean Density Iterations	49.3251
Std Dev. Density Iterations	21.3806

Table 4.5: Density Solver Iteration Statistics

change rate error in accordance to the threshold values for the time step. The average number of iterations to solve for divergence is approximately seven, with a standard deviation of three and a trend of decreasing as the simulation progressed, which can be seen in Table 4.1 and 4.3. Solving for divergence is not expensive computationally and helps lower the computation costs for the constant density solver.

The average volume compression error (kg/m^3) is set to stay below 0.01% * rest density (1000 kg/m^3), which equates to 0.1 kg/m^3. Based on Figure 4.5 and average density error from Table 4.4, the constant density solver is accurately keeping the state at or below this threshold for the majority of the simulation. The outliers shown are during the emission phase with all the emitters on, which caused scenarios where the solver maxed out at 100 iterations and did not completely solve the system for the error threshold. The emission phase used a larger time step, and the spikes

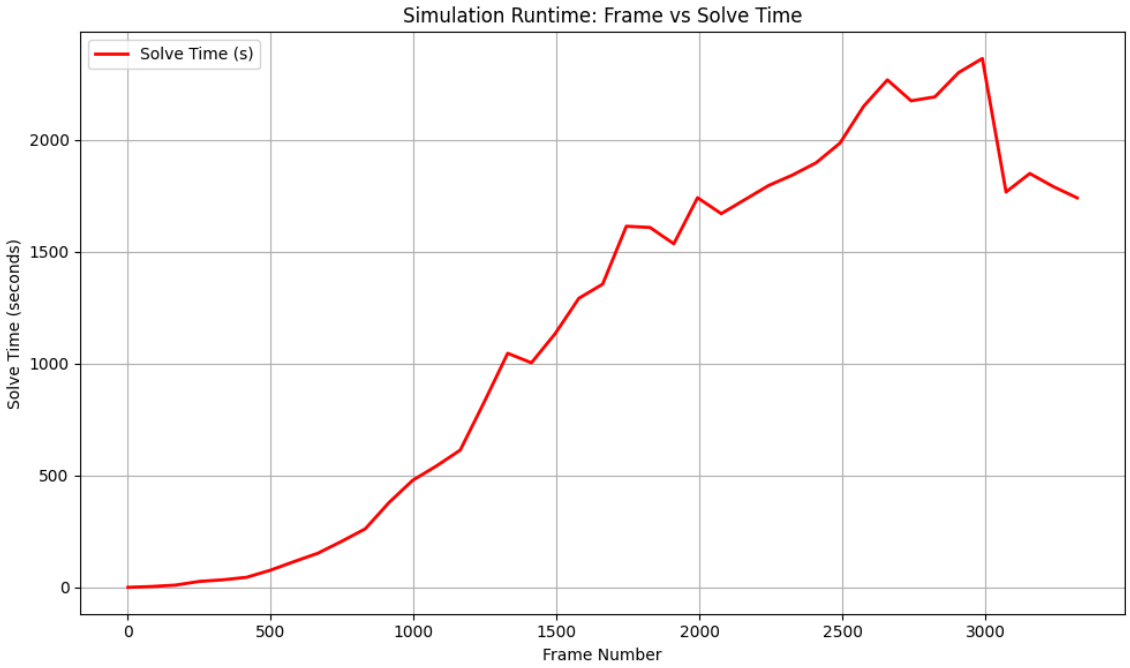

Figure 4.6: Solve Time Graph for Both Solvers

only seemed to occur when a large number of particles were mixing in a close vicinity at once. The constant density solver is more time consuming, where solving took an average iteration of approximately 49 and a standard deviation of 21 seen in Table 4.5. However, as the flow becomes more stable and loses energy, the trends show the number of iterations also begin to drop which is seen in Table 4.1.

Figure 4.6 shows the solve time is approximately based on quadratic scaling, with the data collection keeping the number of CPUs the same, but not the specific type of processors. As the simulation becomes more stable with less iterations, the runtime starts to improve. With the solvers consistently keeping the pressure error within the thresholds, the flow of the particles is noticeably more realistic compared to the original SPH method in Gilligan.

4.2 Fluid Rendering

A volumetric grid is generated using spheres with a radius the size of the weight kernel in place of the particles. This grid is converted into geometry and written as an .obj file via a Python

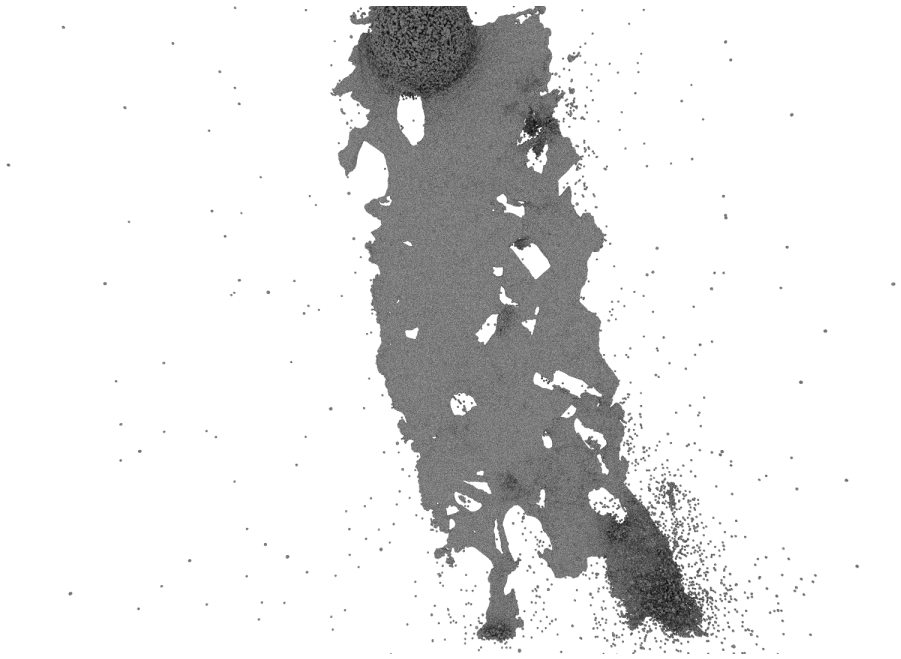

Figure 4.7: Water surface geometry

script. Jobs to generate the volume geometry were used on frames past 2,500 that consisted of 4+ million particles in order to visualize the actual flow and not the emission period. The jobs ran used 20 CPU cores and 30 GB of memory to generate .obj files that average around 20 GB and 2 GB when compressed with 183,701,736 vertices. On average, the performance would range from three to five hours to generate each frame. The geometry can be seen in Figure 4.7, which clearly shows the flyaway particles and isolated particles from the main stream. There is a dense group of particles hitting the end of the streambed at the base of the geometry, which can be seen bouncing back in the opposite direction of the flow. With particles moving realistically, there were less bumps in the geometry which no longer required blurring to hide.

The final render uses Gilligan's GI path tracer, which reads the .obj files for the sky, terrain, and fluid. The fluid geometry applies Gilligan's water volume and ocean surface shader [19] to create the realistic visual seen in Figure 4.8 and 4.9. In Figure 4.9, the flyaway particles are visible and did not create a desirable result. Removing these particles from the .pba files would require the geometry files to be recomputed and the loss of possibly desirable data. Therefore, the cull by neighborhood shader is called during rendering to cull particles based on a neighborhood threshold to avoid this. Culling particles during rendering would be efficient, avoiding re-computations. Simply culling

Figure 4.8: 4 million particles converted from volume to geometry rendered with GI and sky. Shows a close-up of the water surface geometry with a small spray visible.

particles based on height creates issues with the source—particles would splash, become invisible on the way up, and reappear on the way back down. To fix this, a more involved shader was created, cull by neighborhood, to cull particles based on a neighborhood threshold. The shader can significantly lower the amount of particles flying around in the scene while still preserving splashes that occur close to the main bodies of water shown in Figure 4.10 with a threshold of 50. This shader requires reading the pba data for the frame to pass the occupancy grid to the GI path tracer, but does not result in a significant performance add-on. On average, frames after 2,500 used around 100 GB of memory and two hours of computation time with 30 CPUs using OpenMP for parallelization. This shader resulted in shadows, which are more visible in motion, where culled particles close to the terrain geometry still had their shadows visible, as seen in Figure 4.11. To fully get rid of this artifact, the cull by neighborhood logic could be applied during geometry creation. The shader does not fully remove all flyaway particles, but upping the threshold causes splashing particles to phase in and out of visibility and makes the emitter source lose visibility in certain areas, which can be seen in Figure 4.12 with a threshold of 90. The current logic of the shader has limitations that may not be able to adequately cull all the unwanted particles in every scenario.

Figure 4.9: 4 million particles converted from volume to geometry rendered with GI and sky.

Figure 4.10: 4 million particles rendered with water surface using cull by neighborhood shader with threshold 50.

Figure 4.11: 4 million particles rendered showcasing shadows of culled particles.

Figure 4.12: 4 million particles rendered with water surface using cull by neighborhood shader with threshold 90.

Chapter 5

Conclusion and Discussion

The objective of this work was to enhance the stream simulation capabilities in Gilligan and create more realistic water flows and surfaces by replacing the weakly-incompressible SPH method with the fully incompressible DFSPH method. The results show the DFSPH simulation, which includes the improvements of a dynamic time step for stability and more accurate viscosity calculations, creates a much more realistic and stable stream flow compared to the original weakly-incompressible SPH simulation. The simulation is robust and quickly stabilizes during high density errors without ever exploding. The particles are consistently staying incompressible with a divergence-free velocity field leaving artistic parameters as the culprit for visualization quality. The new simulation does not require post-processing effects, such as blurring, to make the fluid geometry look more realistic. Most of the unwanted flyaway particles can easily be culled through a shader call during rendering to avoid recomputing geometry, though shadow artifacts can become present. DFSPH is more computationally expensive than weakly-incompressible SPH, but accurately solving for pressure is what creates better results, and it is not a simple task. These results come from testing the efficacy of DFSPH inside of Gilligan and can be visually improved through artistic parameter manipulation, such as velocity clamps and boundary handling coefficients. Clamping velocity would result in a more stable simulation, with the potential to use a larger time step reducing computations. This would also limit the flyaway particles in the scenes and a more refined visualization could occur by using the cull logic for flyaway particles from the shader during the volume generation to avoid the shadow artifacts.

5.1 Future Works

5.1.1 Performance

Gilligan is often used for prototyping offline simulations and rendering. Hence, the performance is not as optimal as it could be. Future works might improve performance by employing SIMD instructions in the solvers for better parallelization on the hardware level, better neighborhood searches (spatial hashing, oct-trees), and saving calculations such as the weight kernels [6]. It is also possible for DFSPH to be ported to the GPU to create real-time simulations, greatly reducing the time it takes to create the pba files, which is the most time-consuming computation.

5.1.2 Extending SPH

The implemented boundary handling manages particle collisions with geometries but ignores the density at boundaries, which also affect the particles. This can be accounted for in multiple ways, the simplest being the the coating of the streambed in stationary particles, which can result in a large increase in particles for the simulation, increasing the computations. A more complex solution consists of converting the geometry to a level set to use an SDF to determine if the particle has collided and compute the density for the collision point [11], though this may be impractical for certain irregular geometries. Surface tension is not accounted for in this implementation, which can allow for further visual improvements and artistic parameters to alter the result [11].

Bibliography

[1] Mohammad Bagheri, Masoud Mohammadi, and Mehdi Riazi. A review of smoothed particle hydrodynamics. *Computational Particle Mechanics*, 11:1163–1219, 2024.

[2] Jan Bender and Dan Koschier. Divergence-free smoothed particle hydrodynamics. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 1–9, 2015.

[3] Jan Bender and Dan Koschier. Divergence-free sph for incompressible and viscous fluids. *IEEE Transactions on Visualization and Computer Graphics*, 23(3):1193–1206, 2017.

[4] Akio Doi and Akio Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Transactions on Information and Systems*, 74(1):214–224, 1991.

[5] Robert A. Gingold and Joseph J. Monaghan. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 1977.

[6] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel sph implementation on multi-core cpus. *Computer Graphics Forum*, 30(1):99–112, March 2011.

[7] Markus Ihmsen, Jens Cornelis, Barbara Solenthaler, Christoph Horvath, and Matthias Teschner. Implicit incompressible sph. *IEEE Transactions on Visualization and Computer Graphics*, 20(3):426–435, 2014.

[8] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH fluids in computer graphics. In *Eurographics (State of the Art Reports)*, pages 21–42, 2014.

[9] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 143–150, New York, NY, USA, 1986. Association for Computing Machinery.

[10] Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids. In Wenzel Jakob and Enrico Puppo, editors, *Eurographics 2019 - Tutorials*. The Eurographics Association, 2019.

[11] Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. A Survey on SPH Methods in Computer Graphics. *Computer Graphics Forum*, 2022.

[12] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 163–169, New York, NY, USA, 1987. Association for Computing Machinery.

[13] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, 40(2):683–712, 2021.

[14] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543–574, 1992.

[15] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703–1759, 2005.

[16] J. J. Monaghan and R. A. Gingold. Shock simulation by the particle method sph. *Journal of Computational Physics*, 52:374–389, 1983.

[17] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering, fourth edition: From Theory to Implementation*. MIT Press, 2023.

[18] Barbara Solenthaler and Renato Pajarola. Predictive-corrective incompressible sph. *ACM Transactions on Graphics*, 28(3):40:1–40:6, 2009.

[19] Jerry Tessendorf. Gilligan: A prototype framework for simulating and rendering maritime environments, 2017.

[20] Jerry Tessendorf. Visualizing flow conditions using a stream digital twin. Presented at the South Carolina Water Resources Conference, October 19–20, 2022, 2022.

[21] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 9–es, New York, NY, USA, 2005. Association for Computing Machinery.