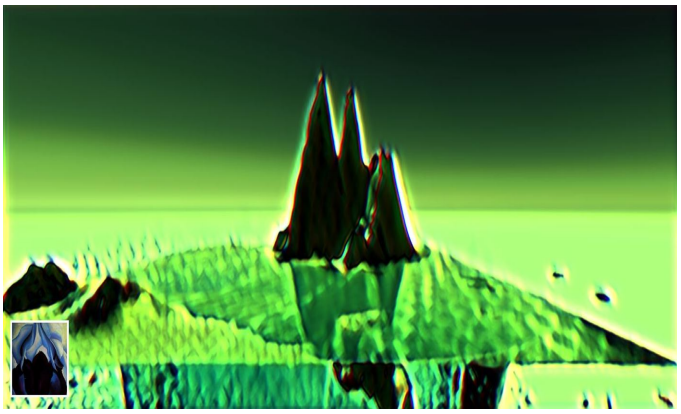
A Style Transfer Guided Generative Architecture for 3D Fluid Mesh and Texture Generation

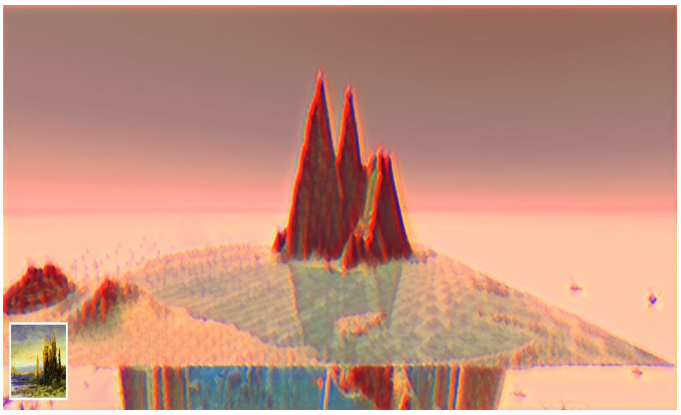
Xi Liu
New York University
New York, USA
xl3504@nyu.edu

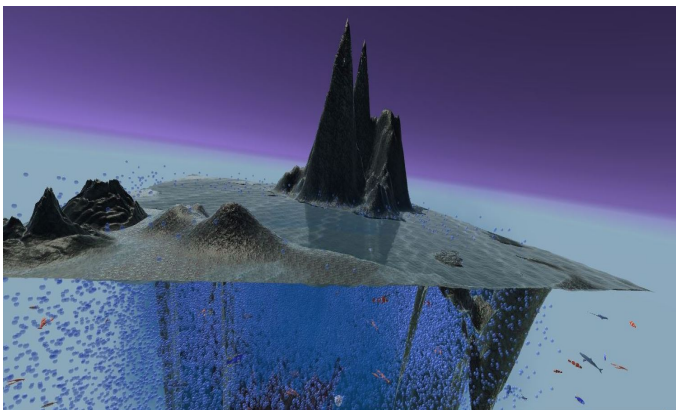
Frances Yuan
New York University
New York, USA
fy576@nyu.edu

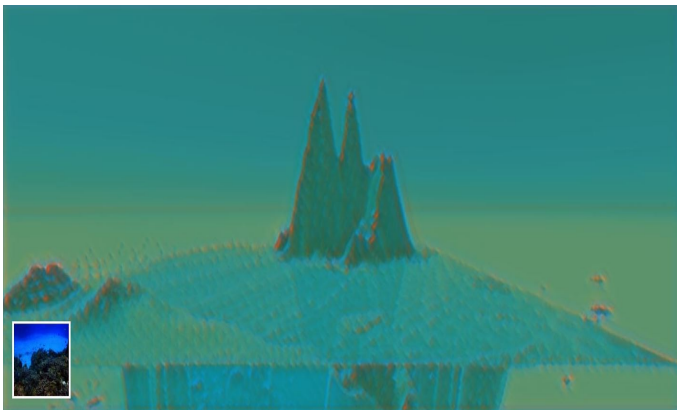
ABSTRACT

Fluid dynamics plays a crucial role in computer vision and underwater exploration. Remote operated vehicles (ROVs), autonomous underwater vehicles (AUVs), and robots operating in fluid environments need to model their interactions with the fluid medium accurately. Understanding the fluid dynamics in such environments is essential for designing efficient and safe robotic systems that can navigate with visual input and perform tasks effectively. In space exploration, fluid dynamics is important in designing spacecraft, propulsion systems, and spacesuits. Space rovers that explore oceans on other celestial bodies, such as Titan or Enceladus, need to have an understanding of the fluid environment they will be operating in to make better visually guided autonomous decisions. Creating physically accurate fluid simulations that are consistent with the scientist's specific need can be challenging. To address this problem, we introduce an approach that employs neural style transfer to generate stylized 3D fluids using a Lagrangian model of flow. This technique offers several advantages over traditional grid-based methods. In particular, particle attributes can be easily transported through the fluid motion, ensuring temporal consistency and improving the overall quality of the result. The method also eliminates the need for computationally expensive velocity field alignment, enabling real-time production workflows. The Lagrangian representation further enhances the scientist's control by enabling multi-fluid stylization and consistent color transfer from input images.









1 INTRODUCTION

Recent activity in geometric deep learning, neural rendering, and algebraic differential geometry has spurred emergence of many neural networks that are suitable for mesh and texture generation, i.e., generative adversarial networks, diffusion networks,

variational autoencoders, neural radiance fields, and transformers. Based on the programmability of the graphics pipeline, the community has established rendering techniques based on pure physical models that can produce photorealistic imagery, but the pure physical approach requires a precise definition of the input parameters such as surface geometry, boundary conditions, material properties, and light. In this task, we use a neural algorithm in tandem with the physical model to perform style transfer on fluids in 3D scenes.

The system can perform style transfer or texture transfer. Given an image of a particular style such as that of Van Gogh's starry night as input, the convolutional neural network would extract a representation of the style of the artwork and reconstruct the 3D scene with the newly transferred style. The reconstructed 3D scene would maintain the basic geometry and hierarchical objects from the original 3D scene, but the texture of the 3D scene will be transformed [15] [43].

2 METHOD

2.1 Content representation

let $\overrightarrow{p}$ and $\overrightarrow{x}$ be the the original image and generated image, P^l and F^l their respective feature representation in layer l. define squared error loss function between two feature representations

$$\mathcal{L}_{content}(\overrightarrow{p}, \overrightarrow{x}, l) = \frac{1}{2} \sum_{i,j} (F^l_{ij} - P^l_{ij})^2$$

derivative of this loss with respect to activations in layer l equals

$$\frac{\partial \mathcal{L}_{content}}{\partial F^l_{ij}} = \begin{cases} (F^l - P^l)_{ij} & \text{if } F^l_{ij} > 0 \\ 0 & \text{if } F^l_{ij} < 0 \end{cases}$$

The content representation in higher layers of the network captures the high-level content regarding objects and their arrangement in the input image. However, it does not significantly limit the precise pixel values of the reconstruction. On the other hand, reconstructions obtained from lower layers merely replicate the exact pixel values of the initial image. As a result, we label the feature responses in the higher layers of the network as the content representation.

2.2 Style representation

To obtain texture information, a feature space is utilized, and the feature correlations are determined by the Gram matrix $G^l \in \mathcal{R}^{N_l \times N_l}$. G^l_{ij} represents the inner product between the vectorized feature maps i and j in layer l.

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}$$

let $\overrightarrow{a}$ and $\overrightarrow{x}$ be original image and generated image, A^l and G^l their respective style representation in layer l. The contribution of layer l to the total loss is

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G^l_{ij} - A^l_{ij}\right)^2$$

total style loss is

$$\mathcal{L}_{style}(\overrightarrow{a}, \overrightarrow{x}) = \sum_{l=0}^{L} w_l E_l$$

w_l are weighting factors of the contribution of each layer to the total loss. The derivative of E_l with respect to activations in layer l can be computed analytically

$$\frac{\partial E_l}{\partial F^l_{ij}} = \begin{cases} \frac{1}{N_l^2 M_l^2}((F^l)^T(G^l - A^l))_{ji} & \text{if } F^l_{ij} > 0 \\ 0 & \text{if } F^l_{ij} < 0 \end{cases}$$

2.3 Style transfer

The task is to transfer the style of an artwork $\overrightarrow{a}$ onto a photograph $\overrightarrow{p}$. To achieve this, we start from a random white noise image, and iteratively modify the pixel values to simultaneously minimize the content representation difference between the generated image and the photograph in one layer, and the style representation difference between the generated image and the painting defined on multiple layers of the convolutional neural network. The loss function we minimize is

$$\mathcal{L}_{total}(\overrightarrow{p}, \overrightarrow{a}, \overrightarrow{x}) = \alpha \mathcal{L}_{content}(\overrightarrow{p}, \overrightarrow{x}) + \beta \mathcal{L}_{style}(\overrightarrow{a}, \overrightarrow{x})$$

α and β are the weighting factors for content and style reconstruction.

2.4 Fluid style transfer

In a Lagrangian 3D fluid, it is beneficial to use particles or fluid parcels for style transfer since attributes are computed on a per particle basis and transported by the particle motion, which provides consistency of the stylized structure across time.

We add semantic structures onto volumetric flow data, based on neural style transfer techniques used in image and mesh stylization. The approach involves modifying 3D density fields using multiple 2D stylizations synthesized by matching features of a pre-trained convolutional neural network for image classification tasks. By using a vast library of patterns and class semantics available from the convolutional neural network, our method enables content-aware flow manipulations that range from low-level features like edges and patterns to high-level features like complex structures and shapes. Our algorithm is physically inspired and computes a velocity field that stylizes a water density with an input target style, yielding results that model the underlying transport phenomena. The method improves temporal consistency by aligning stylization velocities from adjacent frames, allowing control over how smoothly stylized structures change over time. The method is end-to-end differentiable and optimized by gradient descent approaches, enabled by a novel volumetric differentiable fluid renderer tailored for stylization purposes. Our approach captures a wide spectrum of styles and high-level semantics, including patterns, regular structures, turbulence effects, shapes, and textural styles, making it useful for transferring these features onto existing simulations.

A Eulerian-based transport based neural style transfer (TNST) is an algorithm to transfer the image style to a 3D fluid density [8]. The Eulerian view of fluid motion describes the velocity field as a function of space and time with the vector function $v(x, y, z, t)$.

Transport phenomena are irreversible processes from the random motion of molecules in fluids grounded in conservation laws and constitutive equations describing systems turning into their lowest energy state. In neural style transfer (NST), the optimization is performed on individual pixels of the target image; whereas in TNST, the optimization is performed on a velocity field that updates density values. The loss function $\mathcal{L}$ computed from an image classification convolutional neural network defines the velocity field $v : \mathbb{R}^3 \to \mathbb{R}^3$ through the stylization of input density $d : \mathbb{R}^3 \to \mathbb{R}^3$

$$\hat{v} = \arg\min_{v} \sum_{\theta \in \Theta} \mathcal{L}(\mathcal{R}_\theta(\mathcal{T}(d, v)), p)$$

The transport function $\mathcal{T}$ takes a velocity field $\hat{v}$ to advect a given density field d, resulting in a stylized density field $\hat{d} = \mathcal{T}(d, \hat{v})$. The differentiable renderer $\mathcal{R}$ then converts this stylized density field into an image in image-space for a specific view angle θ, resulting in the final image $I = \mathcal{R}_\theta(\hat{d})$. The stylization process includes user-defined parameters represented by p. The velocity field contributions are computed separately for each view, resulting in a 3D volumetric fluid stylization. The authors of the technique divide the velocity field into its irrotational and incompressible parts for optimization.

The loss function is divided into two components: semantic and style losses. This allows for additional control over the stylization given a rendered density field. For style transfer, the input image and user-selected activation layers are considered, while for semantic transfer, a CNN layer with desirable attributes is selected to be transferred to the target stylized fluid. The advected fluid is guaranteed to maintain its original shape and semantics since it is being advected towards a target objective, without needing to match its original content loss as in traditional neural style transfer algorithms. The style loss is given by

$$\mathcal{L}_s(I, p_s) = \sum_{l}^{L} \left(\frac{1}{4C_l^2 (H_l \times W_l)^2} \sum_{m,n}^{C_l} (G^l_{mn}(I) - G^l_{mn}(I_s))^2 \right)$$

The Gram matrix G measures correlations between different filter responses. The Gram matrix is computed for a specific layer l and for two channels, m and n. The calculation involves iterating over all pixels of the flattened 1-D feature map $\hat{F}^l(I)$, where I is the input image. The flattened feature map is used to calculate the correlation between the activations of the two channels, m and n. This process is repeated for all pairs of channels, resulting in the Gram matrix G.

$$G^l_{mn}(I) = \sum_{l}^{H_l \times W_l} \hat{\mathcal{F}}_{mi}(I) \hat{\mathcal{F}}_{ni}(I)$$

The process of extending a single frame stylization over time in a coherent manner can be inaccurate and computationally expensive when using an Eulerian framework. To address this, TNST aligns stylization velocities by recursively advecting them with simulation velocities for a given window size. This recursive approach can be inefficient in terms of both time and memory, particularly when larger window sizes are used to ensure smooth transitions between consecutive frames. As a result of the large memory requirements, this operation may need to be computed on the CPU, which adds further overhead due to the use of expensive data transfer operations.

The forces acting on a moving fluid element consist of body forces and surface forces. The body forces include gravitational, electric, and magnetic forces that act on the volumetric mass of the fluid element. The surface forces act on the surface of the fluid element due to pressure on the surface by outside surrounding fluid, and shear and normal stress distributions acting on the surface [4]. Let ρ be the density, $\mathbf{V} = (u, v, w)$ be the flow velocity, $\frac{D}{Dt} = \frac{\partial}{\partial t} + \nabla\mathbf{V}$ be the substantial derivative, p be the pressure, τ be the shear stress, f be the body force per unit mass, from Newton's second law, we obtain

$$\rho \frac{D\mathbf{V}}{Dt} = -\nabla p + \nabla \tau + \rho f$$

$$\begin{aligned}
\rho \frac{Du}{Dt} &= \frac{\partial(\rho u)}{\partial t} + \nabla(\rho u \mathbf{V}) = \frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} + \frac{\partial(\rho uw)}{\partial z} \\
&= -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \\
\rho \frac{Dv}{Dt} &= \frac{\partial(\rho v)}{\partial t} + \nabla(\rho v \mathbf{V}) = \frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho v^2)}{\partial y} + \frac{\partial(\rho vw)}{\partial z} \\
&= -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \\
\rho \frac{Dw}{Dt} &= \frac{\partial(\rho w)}{\partial t} + \nabla(\rho w \mathbf{V}) = \frac{\partial(\rho w)}{\partial t} + \frac{\partial(\rho uw)}{\partial x} + \frac{\partial(\rho vw)}{\partial y} + \frac{\partial(w^2)}{\partial z} \\
&= -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z
\end{aligned}$$

The Lagrangian view of fluid motion use the concept of fluid particle or fluid element. The equation of motion of i-th element of fluid with density ρ_i, volume Δv_i, center of mass r_i, pressure gradient ∇p, body force f_i is

$$\rho_i \Delta v_i \frac{d^2 r_i}{dt^2} = -\Delta v_i \nabla p + \rho_i \Delta v_i f_i \quad [16]$$

$$\frac{d^2 r_i}{dt^2} = -\frac{1}{\rho_i} \nabla p + f_i$$

Dirac δ distribution

$$\delta(\mathbf{r}) = \begin{cases} \infty & \text{if } \mathbf{r} = \mathbf{0} \\ 0 & \text{otherwise} \end{cases}$$

$$\int \delta(\mathbf{r}) d\mathbf{r} = 1$$

$A(\mathbf{x})$ is a continuous compactly supported function.

The convolution between two functions f and g is defined as $(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$. Let $\varepsilon \in \mathbb{R}^+$, the convolution of $A(\mathbf{x})$ with the Dirac δ distribution is identical to A itself:

$$\begin{aligned}
(A * \delta)(\mathbf{r}) &= \int A(\mathbf{r}') \delta(\mathbf{r} - \mathbf{r}') d\mathbf{r}' \\
&= \lim_{\varepsilon \to 0} \int_{-\infty}^{r-\varepsilon} A(\mathbf{r}') \delta(\mathbf{r} - \mathbf{r}') d\mathbf{r}' \\
&\quad + \int_{r-\varepsilon}^{r+\varepsilon} A(\mathbf{r}') \delta(\mathbf{r} - \mathbf{r}') d\mathbf{r}' \\
&\quad + \int_{r+\varepsilon}^{\infty} A(\mathbf{r}') \delta(\mathbf{r} - \mathbf{r}') d\mathbf{r}'
\end{aligned}$$

$$
\begin{aligned}
&= \lim_{\varepsilon \to 0} \int_{r-\varepsilon}^{r+\varepsilon} A(\mathbf{r}')\delta(\mathbf{r}-\mathbf{r}')d\mathbf{r}' \\
&= A(\mathbf{r}) \int \delta(\mathbf{r}-\mathbf{r}')d\mathbf{r}' \\
&= A(\mathbf{r})
\end{aligned}
$$

Let $d \in \mathbb{Z}, W : \mathbb{R}^d \times \mathbb{R}^+ \to \mathbb{R}$ is kernel function or smoothing kernel

$$A(\mathbf{r}) \approx (A * W)(\mathbf{r}) = \int A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}', h)d\mathbf{r}'$$

h is kernel's smoothing length that controls the amount of smoothing and how strongly the value of A at position $\mathbf{x}$ is influenced by the values in its close proximity.

$$\mathcal{N}(\mathbf{r}; \mu, \sigma^2), \lim_{\sigma \to 0} \mathcal{N}(\mathbf{r}; 0, \sigma^2) = \delta(\mathbf{r})$$

$$\int_{\mathbb{R}^d} W(\mathbf{r}', h)d\mathbf{r}' = 1 \quad \text{(normalization condition)}$$

$$\lim_{h' \to 0} W(\mathbf{r}, h') = \delta(\mathbf{r}) \quad \text{(Dirac } \delta \text{ condition)}$$

$$W(\mathbf{r}, h) = 0 \text{ for } \|\mathbf{r}\| \geq h \quad \text{(compact support condition)}$$

We typically use a distribution that is close to a Gaussian distribution to approximate the Dirac δ distribution
Let W be a function satisfying $\int W(\mathbf{r})d\mathbf{r} = 1$, $\mathbf{r}$ be the center of mass, the smoothed density $\rho_s(\mathbf{r})$ is

$$\rho_s(\mathbf{r}) = \int W(\mathbf{r}-\mathbf{r}')\rho(\mathbf{r}')d\mathbf{r}'$$

The fluid quantities are interpolated and spatial derivatives are approximated with finite number of sample positions of adjacent particles. Let $\mathcal{F}$ be the set containing all point samples, the analytic integral is approximated by a sum over sampling points:

$$
\begin{aligned}
A(\mathbf{r}) &= (A * W)(\mathbf{r}_i) \\
&= \int A(\mathbf{r}')W(\mathbf{r_i}-\mathbf{r}', h)d\mathbf{r}' \\
&\approx \sum_{j \in \mathcal{F}} A_j W(\mathbf{r_i}-\mathbf{r_j}, h)v_j \\
&= \sum_{j \in \mathcal{F}} A_j W(\mathbf{r_i}-\mathbf{r_j}, h)\frac{m_j}{m_j/v_j} \\
&= \sum_{j \in \mathcal{F}} A_j W(\mathbf{r_i}-\mathbf{r_j}, h)\frac{m_j}{\rho_j} \\
&= \sum_{j \in \mathcal{F}} A_j \frac{m_j}{\rho_j} W(\mathbf{r_i}-\mathbf{r_j}, h)
\end{aligned}
$$

all field quantities indexed using a subscript denote the field evaluated at the respective position. $A_j = A(\mathbf{x}_j), W_{ij} = W(\mathbf{x_i}-\mathbf{x_j}, h)$
Let S be the set of particles

Algorithm state equation

for particle$_i \in S$
  find neighbor j
for particle$_i \in S$
  $\rho_i = \sum_j m_i W_{ij}$
  compute p_i **using** ρ_i
for particle$_i \in S$
  $f_i^{pressure} = \frac{m_i}{\rho_i}\nabla p_i$
  $f_i^{viscosity} = m_i v \nabla^2 v_i$
  $f_i^{other} = m_i g$
  $f_i(t) = f_i^{pressure} + f_i^{viscosity} + f_i^{other}$
for particle$_i \in S$
  $v_i(t+\Delta t) = v_i(t) + \Delta t f_i(t)/m_i$
  $x_i(t+\Delta t) = x_i(t) + \Delta t v_i(t+\Delta t)$

The Lagrangian representation relies on particles to carry information such as position, density, and color, while the Eulerian counterpart does not. In neural style transfer, loss functions are computed based on filter activations from pre-trained classification networks, which are designed for image datasets. To use these methods, we need to transfer information between particles and grids, where loss functions and attributes can be updated together. We can draw inspiration from hybrid Lagrangian-Eulerian fluid simulation pipelines, which use both grid-to-particle (I_{g2p}) and particle-to-grid (I_{p2g}) transfers.

$$\lambda^\circ = I_{g2p}(x^\circ, \lambda^\boxplus), \quad \lambda^\boxplus = I_{p2g}(x^\circ, \lambda^\circ, h, x^\boxplus)$$

The attributes λ° and λ are defined on particles and grids, respectively. x° represents the positions of all particles, while x represents the grid nodes where values are transferred. The support size of the particle-to-grid transfer is denoted by h. To transfer values from the grid to particles, we use a regular grid cubic interpolant, and for the particle-to-grid transfer, we use standard radial basis functions. Regular Cartesian grids are beneficial for finding grid vertices near a particle position. To achieve this, we extended a differentiable point cloud projector to work with arbitrary grid resolution, neighborhood size, and custom kernel functions. To compute the grid attribute λ, we sum up the weighted particle contributions from all neighboring particles $j \in \partial\Omega_x$ around a grid node x.

$$
\begin{aligned}
\lambda^\boxplus(x) &= \frac{\sum_{j \in \partial\Omega_x} \lambda_j^\circ w(\|x - x_j^\circ\|, h)}{\sum_{j \in \partial\Omega_x} w(\|x - x_j^\circ\|, h)} \\
\phi(x) &= |x - \overline{x}| - \overline{r} \\
\overline{x} &= \sum_i w_i x_i, \quad \overline{r} = \sum_i w_i r_i \\
w_i &= \frac{k(|x - x_i|/R)}{\sum_j k(|x - x_j|/R)}
\end{aligned}
$$

To perform mesh reconstruction from particles, we need to construct a surface that wraps around the particles given particle positions. The metaball approach is best fit for only a few particles [45]. We instead uses the particle neighborhood information for each cube corner. Let $x \in \mathbb{R}^3$ be a cube corner inside the bounds of the SPH simulation, $\overline{x} \in \mathbb{R}^3$ be the weighted average particle position, $\overline{r} \in \mathbb{R}$ be the weighted average particle radius. The voxel density function $\phi : \mathbb{R}^3 \to \mathbb{R}$ that we used is given in above equation. $k : \mathbb{R} \to \mathbb{R}, k(s) = (1 - s^2)^3$ is the kernel function that we used. $R \in \mathbb{R}$ is the radius of neighborhood around x.
W is a kernel that is commonly used in smoothed particle hydrodynamics simulations The kernel W in a SPH simulation can be represented by the form below [26], [5]. Let $d \in \mathbb{Z}$ be the dimension, $f : \mathbb{R}^d \times \mathbb{R}^+ \to \mathbb{R}$ be a function, h is the kernel's smoothing

length, r is the center of mass, c be a scaling factor

$$W(r,h) = \frac{1}{ch^d}\begin{cases} f(r,h) & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

$$W_{\text{poly6}}(r,h) = \frac{315}{64\pi h^9}\begin{cases} (h^2 - r^2)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

$$W_{\text{cubic}}(r,h) = \begin{cases} \frac{2}{3} - r^2 + \frac{1}{2}r^3 & 0 \le r \le 1 \\ \frac{1}{6}(2-r)^3 & 1 \le r \le 2 \\ 0 & r > 2 \end{cases}$$

Now we have the necessary algebraic structures to convert the loss function for the Eulerian fluid style transfer into a Lagrangian fluid style transfer framework. Let Λ° be a set of Lagrangian attributes, the objective function to optimize in a single frame is

$$\Lambda^\circ = \arg\min_{\Lambda^\circ} \sum_{\theta\in\Theta} \sum_{\lambda^\circ\in\Lambda^\circ} w_{\lambda^\circ} \mathcal{L}(R_\theta(I_{p2g}(x^\circ, \lambda^\circ)), p)$$

w_{λ° are weights for the loss functions accounting for Lagrangian attributes. When particle position x° is used for as the target quantity λ°. The SPH density is used for $I_{p2g}(x^\circ) = \sum_{j\in\partial\Omega_x} m_j w(\|x - x^\circ\|, h)$. The equation $\rho_i = \sum_j m_j W_{ij}$ comes from the assumption that the mass density at a given particle i can be estimated by summing up the masses of all neighboring particles j within a certain radius of influence [24]. The mass density at a particle i is therefore influenced by the masses of all neighboring particles j that are close enough to interact with it. The weight W_{ij} in the formula represents the influence of particle j on the mass density at particle i. It is a function of the distance between particles i and j, and it decreases with increasing distance. By summing up the masses of all nearby particles weighted by the kernel function, we obtain an estimate of the mass density at particle i.

$$\mathcal{L}(\lambda^\circ)_\rho = \left(\sum \Delta\lambda^\circ\right)^2 - \sum \log \|\Delta\lambda^\circ\|_1$$

The first loss function is suitable for volumetric data simulated with grid-based solvers. In order to stylize fluid scenes, we need to optimize a scalar value λ° that is represented by the Lagrangian particles. In most cases, this scalar value corresponds to the density of the fluid, but it can also be the color or emission. To ensure that the amount of fluid is conserved during the stylization process, we use a regularization term that minimizes the total net change in fluid. This helps to prevent the stylization from causing particles to fade out unnecessarily and keeps the changes in the fluid non-zero. Additionally, we use cross-entropy loss ($-\sum_x p(x)\log p(x)$) to further minimize the impact of the regularization term. Figure 5 provides a visual representation of how different weights for the regularizer can affect the stylization results.

$$\mathcal{L}(x^\circ)_{\Delta x} = \|I_{p2g}(x^\circ) - \rho_0^{\boxplus}\|_2^2$$

The second loss functions is suitable for particle-based or hybrid fluid solvers. In our approach, we use optimized Lagrangian attributes to define the displacements of particle positions for generating stylizations. However, modifying particle displacements can result in cluttering or regions with insufficient particles, which can negatively impact the stylization quality. To address this, we introduce a regularization term that penalizes irregular distribution of particle positions. This term is defined in above equation, where $\rho_0^{\boxplus}$ represents the rest density for cells that contain particles and is zero for cells without particles. Both loss functions are incarnations of mass conservation property which is enforced by separating the irrotational and incompressible terms to be optimized independently.

When working with grid-based simulations as input, we need to sample and simulate particles. To reduce computational cost, we use a sparse representation that involves only one particle per voxel, as opposed to the 8 particles per voxel used in hybrid liquid simulations for momentum conservation. However, using a low number of particles and a position integration algorithm can lead to irregularly distributed particles, resulting in overly dense or void regions in the rendered image. To address this, we solve an optimization problem

$$\hat{x}^\circ, \hat{\rho}^\circ = \arg\min_{x^\circ,\rho^\circ} \sum_t \|I_{p2g}(x_t^\circ, \rho_t^\circ) - \rho_t^{\boxplus}\|_2^2$$

which involves minimizing the difference between the sampled particle densities and the rest densities over time. However, this optimization problem is under-constrained and has a time-varying objective term, making it challenging to optimize jointly for both particle positions and densities. Therefore, we use a heuristic approach by dividing the problem into two steps: position optimization and multi-scale density update. In the position optimization step, we use the above as the objective to minimize the irregular distribution of particle positions.

3 EXPERIMENTS

3.1 Style transfer

We experiment with style transfer models from two influential papers in the field: “Image Style Transfer Using Convolutional Neural Networks" [15] and “Photorealistic Style Transfer via Wavelet Transforms" [42]. There are many open-source code implementations available for both papers, and we adopt the following two GitHub repositories in our study.

- https://github.com/gordicaleksa/pytorch-neural-style-transfer
- https://github.com/ptran1203/photorealistic_style_transfer

We leverage Barracuda [40], a neural inference engine developed by the Unity Labs, to import our pre-trained style transfer models from [15] into a Unity project. Barracuda integrates neural networks into Unity’s rendering loop, making it an ideal tool for our purposes. For more information about Barracuda, please refer to their GitHub repository at https://github.com/UnityLabs/barracuda-style-transfer. Figures through out this paper are screenshots from the Unity project with this model.

We have only implemented model from [42] with static images, but because Barracuda works by taking the camera’s rendered image from each frame, using it as input to the style transfer model, and displaying the output to the screen [40], similar results should be expected in Unity. Results for [42] are at the end of the paper in appendix.

3.2 Fluid style transfer

Below is a compute shader written in High Level Shading Language (HLSL) that takes in two textures, the camera depth texture and

the camera motion vector texture, and outputs a target stylized depth + motion texture. The shader calculates the stylized depth and motion vectors for each pixel in the output texture based on the depth and motion vector values in the input textures.
The compute shader starts by defining the input textures and output texture as variables and setting the thread group size using the numthreads attribute. The main function of the shader is StyleDepthMotion, which takes in the id of the current dispatch thread.
The function first checks if the current dispatch thread's coordinates are within the frame dimensions defined in the input variables _FrameWidth and _FrameHeight. If the current thread is outside the frame dimensions, it returns and does not perform any further calculations.
Next, the function reads the motion vector and depth values of the current pixel from the input textures using the coord variable. It then performs calculations to find a new depth and motion vector that represent the stylized depth and motion for the current pixel. The shader checks whether the current pixel's depth value is close to 1 (which represents the sky in Unity's depth texture) and adjusts the halo size accordingly. Then, the shader searches for neighboring pixels with a minimum depth that is significantly closer to the current pixel's depth value. If such a pixel is found, the function uses its depth and motion vector values instead of the original values for the current pixel. Finally, the shader writes the new motion vector and depth values to the output texture at the current pixel's coordinates. This shader is a part of the rendering pipeline that applies stylized depth and motion effects to a scene.

```
#pragma kernel StyleDepthMotion
#include "UnityCG.cginc"
Texture2D _CameraDepthTexture; // Unity depth texture
SamplerState sampler_CameraDepthTexture;
Texture2D _CameraMotionVectorsTexture; // Unity motion texture
SamplerState sampler_CameraMotionVectorsTexture;
RWTexture2D<float4> _StyleDepthMotionTex; // Target stylized depth
      + motion texture
int _FrameWidth, _FrameHeight, _SkyHaloSize, _HaloSize;
[numthreads(8, 8, 1)]
void StyleDepthMotion(uint3 id : SV_DispatchThreadID)
    if (id.x >= _FrameWidth || id.y >= _FrameHeight) return;
    int2 coord = id.xy;
    float2 mv1 = _CameraMotionVectorsTexture[coord].rg, newMV =
          mv1;
    float depth1 = Linear01Depth(UNITY_SAMPLE_DEPTH(
          _CameraDepthTexture[coord])), newDepth = depth1;
    int haloSearch = _HaloSize;
    if(depth1 >= 0.999) haloSearch = _SkyHaloSize;
    if(haloSearch > 0)
        float minDepth = depth1;
        float2 minDepthMV = mv1;
        int minDepthCount = 1;
        int2 newDepthCoord = coord;
        float newDepthDist = 0.0;
        for (int x = -haloSearch; x <= haloSearch; ++x)
            for (int y = -haloSearch; y <= haloSearch; ++y)
                if(length(int2(x, y)) > haloSearch || (x == 0 && y
                      ==0))// || (x % 2) ^ (y % 2) == true)
                    continue;
                int2 tempCoord = coord + int2(x, y);
                float depth2 = Linear01Depth(UNITY_SAMPLE_DEPTH(
                      _CameraDepthTexture[tempCoord]));
                // New min depth surface found
                if (minDepth - depth2 > (minDepth + depth2) / 2.0f
                      / 15.0f)
                    newDepthCoord = tempCoord;
                    newDepthDist = length(int2(x, y));
                    minDepth = depth2;
                    minDepthMV = _CameraMotionVectorsTexture[
                          tempCoord].rg;
                    minDepthCount = 1;
                // Same min depth surface
                if (abs(minDepth - depth2) < (minDepth + depth2) /
                      2.0f / 15.0f)
                    float tempDist = length(int2(x, y));
                    if (tempDist < newDepthDist)
                        newDepthCoord = tempCoord;
                        newDepthDist = tempDist;
                    ++minDepthCount;
                    minDepth = depth2;
            // If a closer depth with significant enough amount of
                  pixel was found in neighbourhood
            if(minDepthCount > 0 && abs(minDepth - depth1) > (minDepth
                  + depth1) / 2.0f / 15.0f)
                newDepth = minDepth;
                //newMV = minDepthMV;
                newMV = _CameraMotionVectorsTexture[newDepthCoord].rg;
    _StyleDepthMotionTex[id.xy] = float4(newMV.xy, newDepth, 0.0);
```

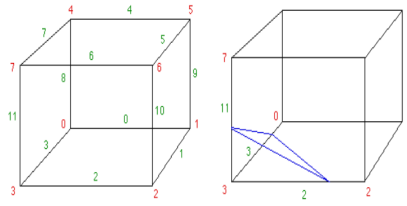


Figure 1: marching cubes

After obtaining the voxel density values from particle positions, we use the marching cubes algorithm to construct the stylized mesh. The marching cubes algorithm locates the surface, creates triangles, and compute the normals of the surface for each vertex of the triangles.
For each cube, the algorithm computes the intersection representation between the cube and the surface. The surface intersections are computed by comparing the voxel value at the vertex with the isovalue of the surface. If the voxel value at the vertex is below the isovalue, we assign a one to the cube's vertex. If the voxel value at the vertex is greater than or equal to the isovalue, we assign a zero to the cube's vertex.

Algorithm cube index

```
cube_index = 0;
for(int i = 0; i < 8; ++i)
    if(cube_corner_density[i] < isolevel)
        cube_index |= 1 << i;
```

Consider the configuration in figure 1 [7], in which green labels denote edge indices and red labels denote vertex indices, suppose the voxel value at vertex 3 is less than the isovalue and every other vertices have a voxel value greater than the isovalue. By algorithm cube index, we obtain a $cube\_index = (0|(1 << 3))_2 = (1000)_2 = (8)_{10}$. Then, we use the $cube\_index$ as the array index of $edge\_table$, finding that $edge\_table[8] = (80c)_{16} = (1000\,0000\,1100)_2$, since bits at position 2, 3, 11 are ones, we find that the surface intersects with cube at the edges 2, 3, and 11.
We use linear interpolation to compute the vertex positions that are at the intersecting edges. Let $i \in \mathbb{R}$ be the isovalue, $p_1, p_2 \in \mathbb{R}^3$ be vertices of a intersecting edge, v_1, v_2 be voxel values at each vertex, then the vertex position $p \in \mathbb{R}^3$ at the intersecting edge is

$$p = p_1 + \frac{i - v_1}{v_2 - v_1}(p_2 - p_1)$$

To find the triangle facets from vertex positions of intersecting edges, we use the same $cube\_index$ as the array index of $triangle\_table$, we find $triangle\_table[8] =$

{3, 11, 2, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1}. By algorithm triangle construction, we construct the triangle from vertices 3, 11, and 2.

Algorithm triangle construction

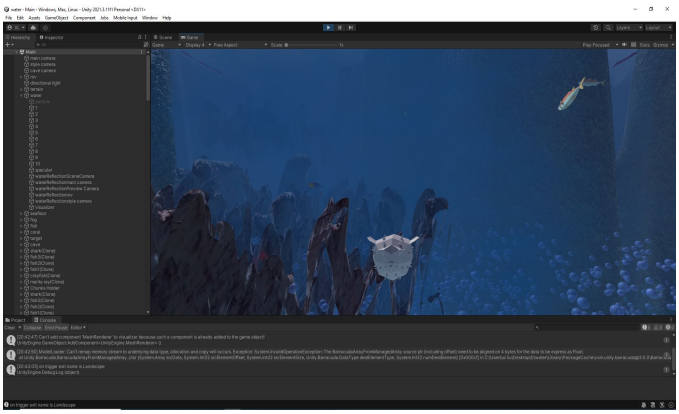
```
for(int i = 0; triangulation[cube_index][i] != -1; i += 3)
    triangle a;
    a.vertex_a = vertex_list[triangle_table[cube_index][i]];
    a.vertex_b = vertex_list[triangle_table[cube_index][i + 1]];
    a.vertex_c = vertex_list[triangle_table[cube_index][i + 2]];
    triangles.append(a);
```

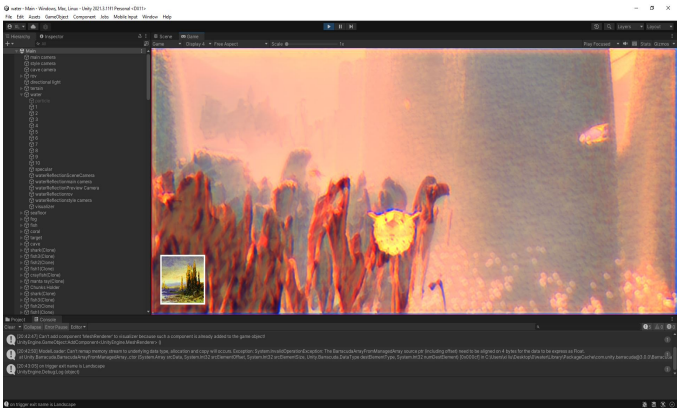
4 RESULTS AND CONCLUSIONS

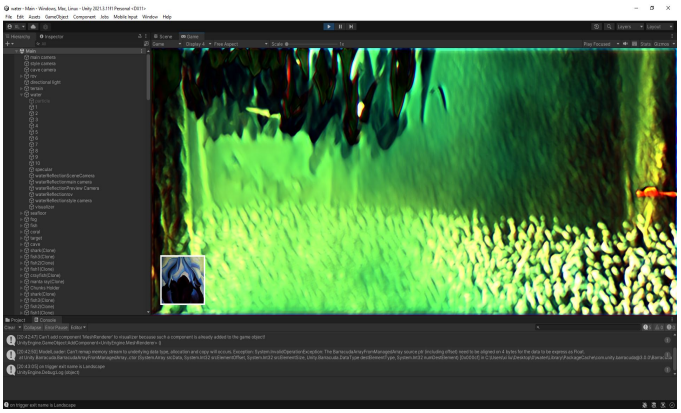
The Lagrangian approach we have presented for neural flow stylization offers several benefits such as improved temporal coherence, faster stylization, and better directability. One of the key advantages of the method is that it can be used with different fluid solvers, including grid, particle, and hybrid solvers. We have also introduced a grid-to-particle transfer strategy to efficiently update attributes and gradients, and a re-simulation approach that can be applied to both grid and particle representations. This method's generality makes it easy to integrate into existing content production workflows.

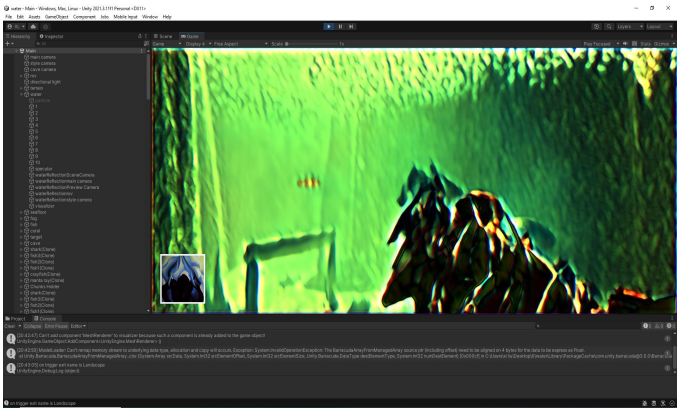
However, there are some limitations to our approach. For example, we use a simple differentiable renderer for liquids, which may not be suitable for all scenarios. A dedicated differentiable renderer for liquids would improve the results and support a wider range of liquid simulation setups. We have not tested the method on large-scale simulations typically used in production settings, and larger scenes may pose challenges with respect to artist control of the stylization. While our method can handle up to 10^6 particles, larger scenes are restricted by available memory.

Overall, the method enables novel effects and a high degree of directability, making flow stylization more practical for production workflows. Our repository is currently at https://github.com/xi-liu-cs/style-transfer.



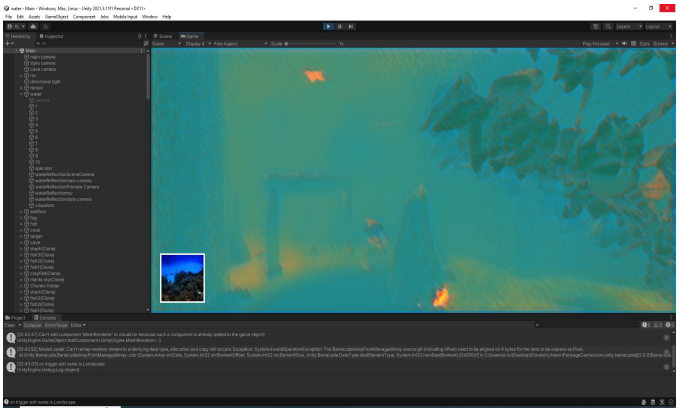
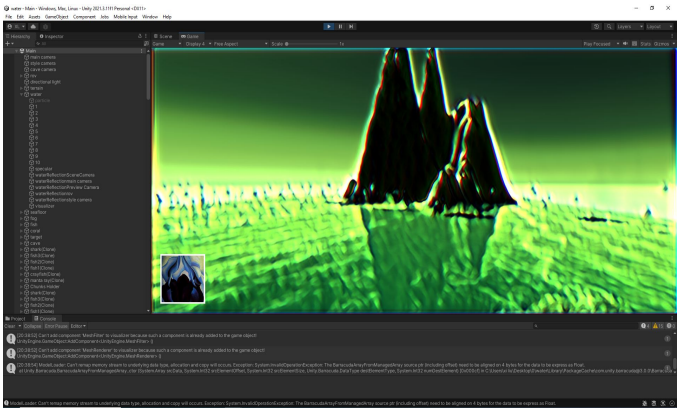
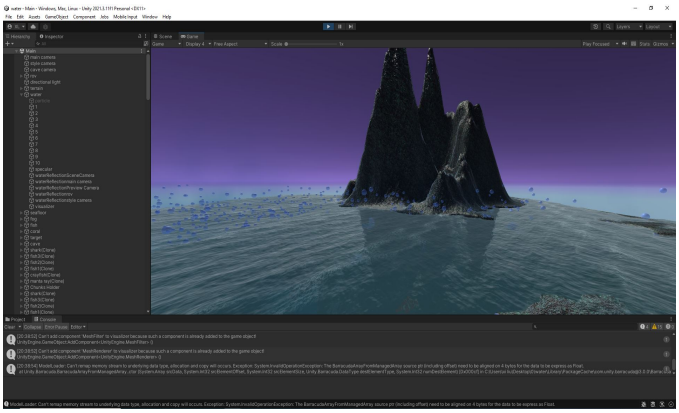
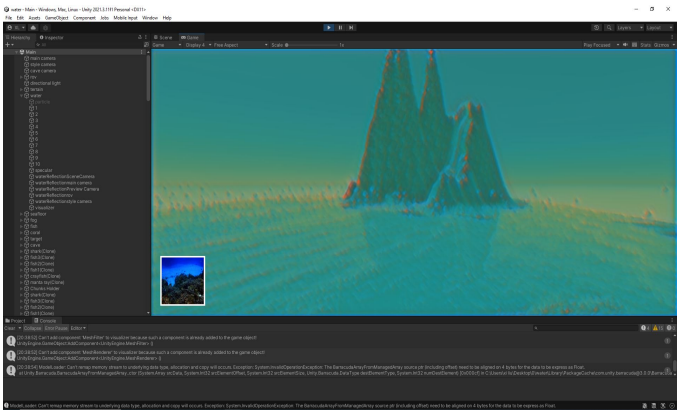
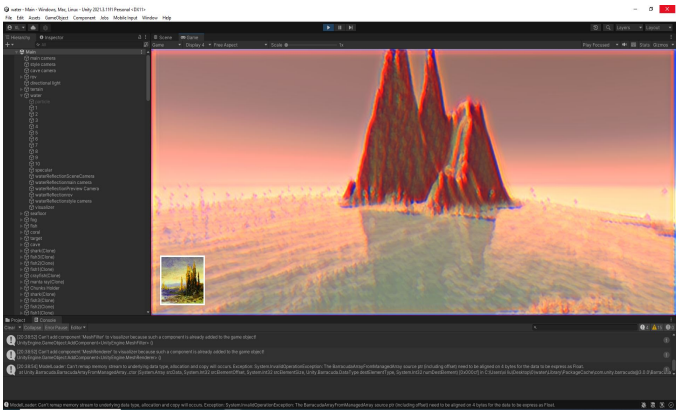






5 DISCUSSIONS AND RELATED WORK

Fluid neural style transfer (FNST) is a variant of neural style transfer (NST) that is specifically designed for stylizing fluid simulations. Unlike traditional NST, which operates on static images, FNST operates on the dynamics of a fluid simulation represented as a sequence of images. The main difference between NST and FNST is the way they handle the dynamics of the input data. NST operates on a static image, while FNST operates on a sequence of images that represent the fluid simulation. This requires additional considerations for temporal consistency and efficient computation. FNST also typically uses a Lagrangian viewpoint, which enables the attributes of the fluid to be stored on particles and transported by their motion, improving temporal consistency and overall quality. Our method is mainly for performing fluid style transfer for Lagrangian model of the fluid flow. There are also fluid style transfer implementations for Eulerian model of the fluid flow [22]. In

the Lagrangian approach, each fluid particle has a unique trajectory, and its properties, such as velocity and density, are tracked along that trajectory. This approach is useful for simulating fluid flows with high particle counts or complex interactions, such as turbulent flows. However, the Lagrangian approach can be computationally expensive and may require high memory usage. In contrast, the Eulerian approach represents the fluid flow as a field that describes the properties of the fluid at each point in space. This approach is useful for simulating fluid flows with regular or uniform characteristics, such as laminar flows or steady-state flows. The Eulerian approach is generally more computationally efficient than the Lagrangian approach and requires less memory.

6 CHALLENGES

Currently the style transfer, smoothed particle hydrodynamics simulation, and the marching cubes voxel density reconstruction pipeline is implemented in the compute shader. For high fidelity simulation and details in the fluid mesh, we need a high amount of particles. When the particle count are over 10^6, the simulation becomes too slow to be run. Also, for many cases the triangles on the particles are only partially constructed, this might be due to too much computation is assigned for each frame and the GPU is not able to finish the computations. Style transfer requires computationally intensive operations, such as convolutions and matrix multiplications, which can be slow to perform on a GPU. Style transfer training involves loading and storing large amounts of data, such as the input image, style image, and intermediate feature maps, which often quickly exhaust the available memory on a GPU. Out-of-memory errors was often encountered.

7 FUTURE SCOPE

The current in-simulation implementation of neural style transfer is not fully detailed. The style transferred mesh is blurred to some extent. For the future, the realism of the converted mesh can be improved by using multiple style references instead of using only one style reference, controlling the level of stylization by adjusting the amount of style transfer applied to the image can help to balance between the stylized and original image, preserving content features by preserving certain features of the original image, such

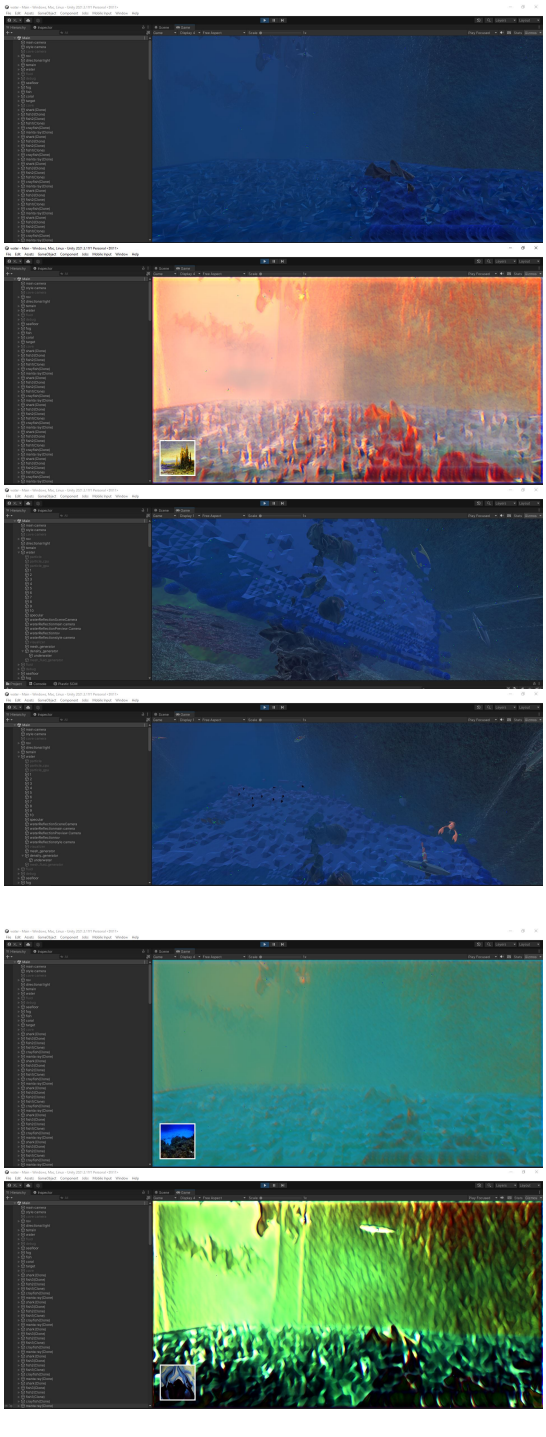


Figure 2: style transferred voxel density fluid mesh

as edges or texture details, the stylized image can be made to look more realistic. We could also use high-resolution images by working with high-resolution images can help to maintain the details and sharpness of the original image, resulting in a more realistic output, fine-tune the style transfer model on a specific dataset or

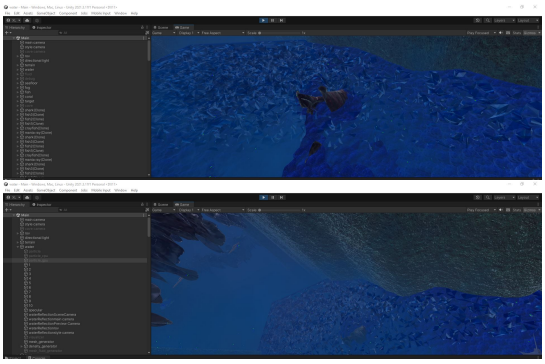
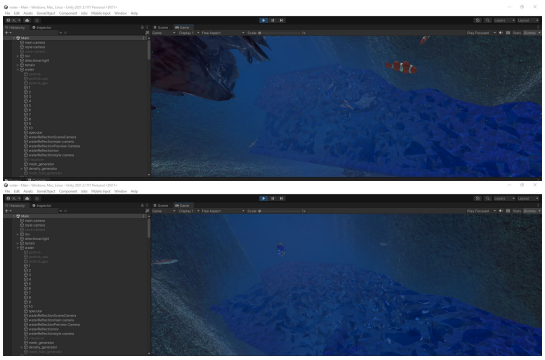


Figure 3: surface reconstruction from particle attributes underwater



image, the model can learn to produce more realistic stylizations, and then use post-processing techniques such as noise reduction, color correction, and sharpening to enhance the realism of a neural style transferred image.

REFERENCES

[1] . [n. d.]. In-Game Style Transfer Tutorial Leveraging Unity* (Part 3). https://www.intel.com/content/www/us/en/developer/articles/training/in-game-style-transfer-leveraging-unity-part-3.html.

[2] . [n. d.]. Pytorch neural style transfer. https://github.com/gordicaleksa/pytorch-neural-style-transfer.

[3] . [n. d.]. Pytorch neural style tutorial. https://github.com/pytorch/tutorials/blob/main/advanced_source/neural_style_tutorial.py.

[4] John David Anderson. 2010. *Computational Fluid Dynamics: The basics with applications.* McGraw-Hill.

[5] Bindel. 2011. 1 deriving SPH - Cornell University. https://www.cs.cornell.edu/~bindel/class/cs5220-f11/code/sph-derive.pdf

[6] James Blinn. 1982. http://papers.cumincad.org/data/works/att/6094.content.pdf

[7] Paul Bourke. 1994. Polygonising a scalar field. http://paulbourke.net/geometry/polygonise/

[8] Byungsoo Kim. [n. d.]. Lagrangian Neural Style Transfer for Fluids. https://arxiv.org/pdf/2005.00803.pdf.

[9] Akash Chaudhary, Rajat Mishra, Bharath Kalyan, and Mandar Chitre. 2021. Development of an Underwater Simulator using Unity3D and Robot Operating System. In *OCEANS 2021 MTS/IEEE.* IEEE. https://github.com/org-arl/UWRoboticsSimulator

[10] Dongdong Chen, Lu Yuan, Jing Liao, Nenghai Yu, and Gang Hua. 2018. Stereoscopic Neural Style Transfer. https://doi.org/10.48550/ARXIV.1802.10591

[11] cj-mills. 2022. End-to-End-In-Game-Style-Transfer-Tutorial-Intel. https://github.com/cj-mills/End-to-End-In-Game-Style-Transfer-Tutorial-Intel.

[12] "cj mills". 2022. In-Game Style Transfer Tutorial Leveraging Unity Barracuda. (2022). https://github.com/cj-mills/End-to-End-In-Game-Style-Transfer-Tutorial-Intel

[13] Thomas Deliot, Florent Guinier, , and Kenneth Vanhoey. 2020. Real-time style transfer in Unity using deep neural networks. (2020).

https://blog.unity.com/engine-platform/real-time-style-transfer-in-unity-using-deep-neural-networks
[14] Martha W Evans, Francis H Harlow, and Eleazer Bromberg. 1957. *The particle-in-cell method for hydrodynamic calculations.* Technical Report. LOS ALAMOS NATIONAL LAB NM. https://apps.dtic.mil/sti/citations/ADA384618
[15] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2016. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2414–2423.
[16] R. A. Gingold and J. J. Monaghan. 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. 181 (Nov. 1977), 375–389. https://doi.org/10.1093/mnras/181.3.375
[17] "Gornhoth" and "jschatteiner". 2022. Unity Smoothed Particle Hydrodynamics. (2022). https://github.com/Gornhoth/Unity-Smoothed-Particle-Hydrodynamics
[18] Francis H Harlow and Billy D Meixner. 1961. *The particle-and-force computing method for fluid dynamics.* Technical Report. LOS ALAMOS NATIONAL LAB NM. https://apps.dtic.mil/sti/citations/ADA384961
[19] Lukas Höllein, Justin Johnson, and Matthias Nießner. 2021. StyleMesh: Style Transfer for Indoor 3D Scene Reconstructions. https://doi.org/10.48550/ARXIV.2112.01530
[20] Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, Yizhou Yu, and Mingli Song. 2019. Neural style transfer: A review. *IEEE transactions on visualization and computer graphics* 26, 11 (2019), 3365–3385.
[21] Tero Karras, Samuli Laine, and Timo Aila. 2018. A Style-Based Generator Architecture for Generative Adversarial Networks. *CoRR* abs/1812.04948 (2018). arXiv:1812.04948 http://arxiv.org/abs/1812.04948
[22] Byungsoo Kim, Vinicius C. Azevedo, Markus Gross, and Barbara Solenthaler. 2019. Transport-based neural style transfer for smoke simulations. https://arxiv.org/abs/1905.07442
[23] Doyub Kim. 2017. Fluid Engine Development. (2017). https://github.com/doyubkim/fluid-engine-dev
[24] Dan Koschier. 2022. https://cg.informatik.uni-freiburg.de/publications/2022_CGF_SPH_survey.pdf
[25] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. 2017. Universal style transfer via feature transforms. *Advances in neural information processing systems* 30 (2017).
[26] Matthias Müller. 2003. https://matthias-research.github.io/pages/publications/sca03.pdf
[27] Ken Perlin and Luiz Velho. 1995. Live paint: Painting with procedural multiscale textures. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques.* 153–160.
[28] Peter Dravecky. 2022. Style transfer in unity neural post processing effect with a transition shader. https://www.artstation.com/artwork/r9XaGa.
[29] W. T. Reeves. 1983. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Graph.* 2, 2 (apr 1983), 91–108. https://doi.org/10.1145/357318.357320
[30] Craig Reynolds. 2006. Big Fast Crowds on PS3. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames* (Boston, Massachusetts) *(Sandbox '06).* Association for Computing Machinery, New York, NY, USA, 113–121. https://doi.org/10.1145/1183316.1183333
[31] Craig W. Reynolds. 1982. Computer Animation with Scripts and Actors. *SIGGRAPH Comput. Graph.* 16, 3 (jul 1982), 289–296. https://doi.org/10.1145/965145.801293
[32] Craig W. Reynolds. 1982. Computer Animation with Scripts and Actors. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques* (Boston, Massachusetts, USA) *(SIGGRAPH '82).* Association for Computing Machinery, New York, NY, USA, 289–296. https://doi.org/10.1145/800064.801293
[33] Craig W. Reynolds. 1987. Flocks, Herds and Schools: A Distributed Behavioral Model. *SIGGRAPH Comput. Graph.* 21, 4 (aug 1987), 25–34. https://doi.org/10.1145/37402.37406
[34] Craig W. Reynolds. 1987. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87).* Association for Computing Machinery, New York, NY, USA, 25–34. https://doi.org/10.1145/37401.37406
[35] Craig W. Reynolds. 1993. An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion. In *Proceedings of the Second International Conference on From Animals to Animats 2: Simulation of Adaptive Behavior: Simulation of Adaptive Behavior* (Honolulu, Hawaii, USA). MIT Press, Cambridge, MA, USA, 384–392. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=84701a575dda6289df142a7a1c051248230d0ca3
[36] Craig W Reynolds et al. 1999. Steering behaviors for autonomous characters. In *Game developers conference*, Vol. 1999. Citeseer, 763–782. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9d19157fa8da0a7d216f44d6a45a73b59b6da23f
[37] Wei Shao and Demetri Terzopoulos. 2005. Autonomous Pedestrians. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Los Angeles, California) *(SCA '05).* Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/1073368.1073371
[38] "SturdySpoon" and Santiago Castro. 2020. Unity Movement AI. (2020). https://github.com/sturdyspoon/unity-movement-ai
[39] Unity Technologies. 2022. Making a New Learning Environment. (2022). https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Learning-Environment-Create-New.md
[40] Kenneth Vanhoey Thomas Deliot, Florent Guinier. 2020. Real-time style transfer in Unity using deep neural networks. https://blog.unity.com/technology/real-time-style-transfer-in-unity-using-deep-neural-networks.
[41] Tamás Vicsek, András Czirók, Eshel Ben-Jacob, Inon Cohen, and Ofer Shochet. 1995. Novel Type of Phase Transition in a System of Self-Driven Particles. *Phys. Rev. Lett.* 75 (Aug 1995), 1226–1229. Issue 6. https://doi.org/10.1103/PhysRevLett.75.1226
[42] Jaejun Yoo, Youngjung Uh, Sanghyuk Chun, Byeongkyu Kang, and Jung-Woo Ha. 2019. Photorealistic style transfer via wavelet transforms. In *Proceedings of the IEEE/CVF International Conference on Computer Vision.* 9036–9045.
[43] Thomas Deliot Zhixiang. 2022. Barracuda Style Transfer. https://github.com/UnityLabs/barracuda-style-transfer.
[44] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. 2017. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *CoRR* abs/1703.10593 (2017). arXiv:1703.10593 http://arxiv.org/abs/1703.10593
[45] Yongning Zhu. 2005. Animating Sand as a Fluid. https://www.cs.ubc.ca/~rbridson/docs/zhu-siggraph05-sandfluid.pdf

Figure 5: Photorealistic Style Transfer via Wavelet Transforms [42] results

APPENDIX

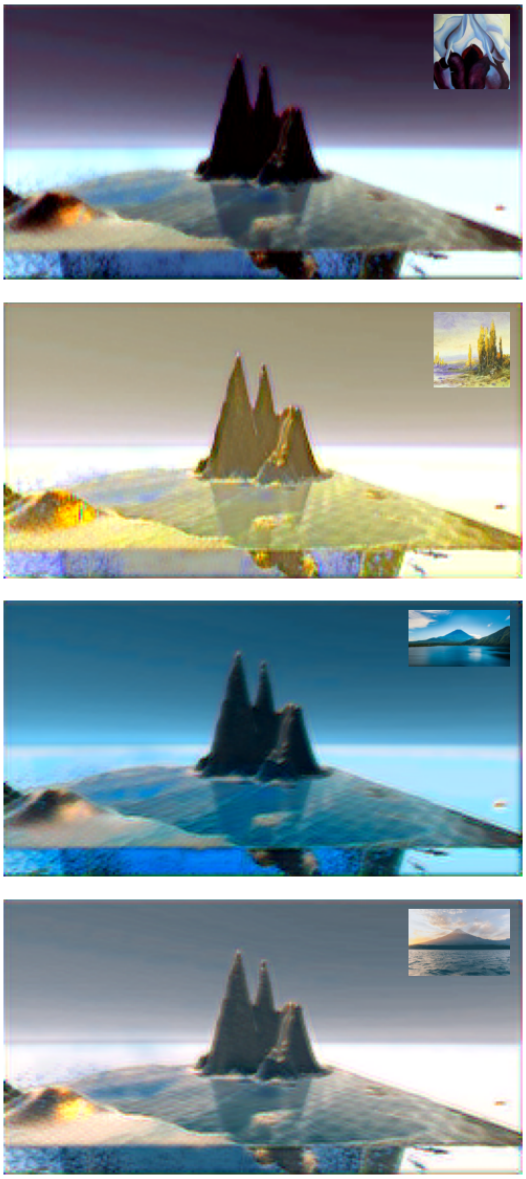


Figure 4: Photorealistic Style Transfer via Wavelet Transforms [42] results