Screen-space 3D fluid rendering in OpenGL

Matteo Princisgh

1 Introduction

The aim of this project is to implement a real-time graphics pipeline capable of visualizing a particle-based fluid simulation implemented in CUDA.
The CUDA simulation, which runs independently on a separate thread, writes the particle data to a shared Vertex Buffer Object, which is used to render the particles in the first steps of our pipeline. The `Particle` type is a `struct{float3, float3, float}` which contains, respectively, the position of a particle, its velocity, and its density.

Typically, there are three main techniques used for rendering fluids in real-time:

- **Marching cubes**: by Lorensen et al. [1], is an algorithm used to create a triangular mesh of the fluid volume. While visually accurate, this approach requires reconstructing the entire fluid surface, which can be computationally expensive for dynamic simulations.
- **Raymarching**: which performs raycasting from the projection plane, and performs regular sampling of the fluid density field. This approach easily allows us to compute multiple refractions along the view-ray at the expense of a high computational cost, making it difficult to optimize and scale for real-time applications.
- **Screen-space techniques**: there are several, but for this project we used some of the ideas proposed by Laan et al. in [2] and by S.Green in [3]. These methods operate primarily in screen space, making them particularly efficient for real-time applications as they focus computational resources only on visible fluid surfaces and scale well with scene complexity.

For this project, we chose to implement screen-space techniques for their balance between visual quality and performance. This approach allows us to achieve convincing fluid rendering with realistic refraction and reflection effects while maintaining high frame rates, even with a large number of particles. All the figures in this report were taken by simulating 50K particles, but a PC equipped with an i7-9700K and an RTX2080 can handle up to 150K particles while maintaining a frame rate of 60 FPS.

2 Particle rendering

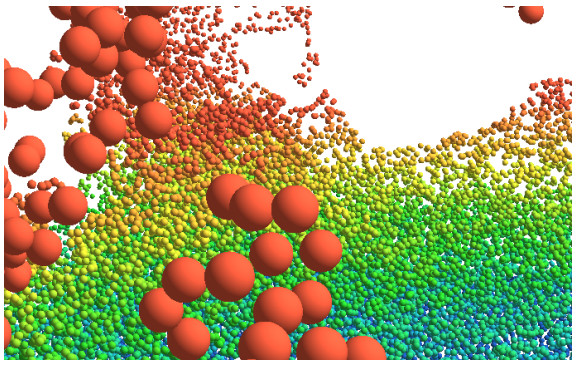


Figure 1: Closeup of the shaded particles.

The first step of our pipeline directly draws the particle VBO using `glDrawArrays()`, which renders each particle as a point sprite. Next, in the vertex shader, we compute the view and projection

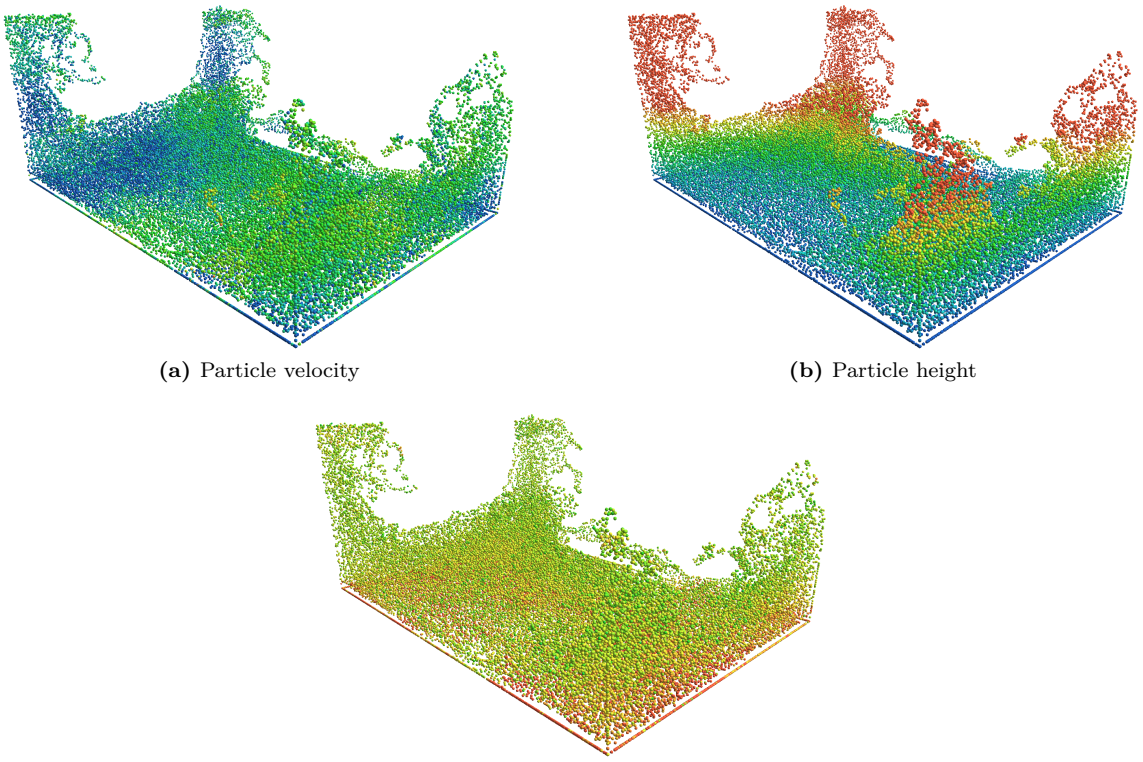
transforms and set the `gl_PointSize` attribute to be proportional to the distance from the projection plane, ensuring that particles further away from the camera appear smaller.
The point sprites drawn on screen are squared, and in order to draw a circle, we need to discard those fragments whose distance from the center is greater than one.
To shade them as if they were spheres, we can use a simple trick to compute their normal:

$$\begin{aligned} p &= \texttt{gl\_PointCoord} \\ N &= \begin{pmatrix} p^{\{x\}} & p^{\{y\}} & 1 - \|p\|_2 \end{pmatrix} \end{aligned} \tag{1}$$

Then with the normal, we can use various shading techniques to have a material-light interaction that gives the illusion of having a sphere in the simulation, as shown in Fig. 1. In our case, we used the Lambertian diffusion lighting model with the Phong ambient component (without the specular highlights); the final color is given by the equation:

$$\texttt{FragColor} = C_{particle} \odot (\alpha \cdot (N \cdot L)) \tag{2}$$

Where a is the ambient component, N the sphere normal, and L the direction of incident light.



(a) Particle velocity

(b) Particle height

(c) Particle density

Figure 2: Visualization of the `Particle` attributes.

Finally, we visualized the particle attributes (i.e. velocity, position, and density) with three fragment subroutines that simply assign a color using a linear interpolation between two extreme colors (red and blue), as shown in Fig.2. Note that this interpolation is performed in the HSV color space to have more eye-pleasing colors.

3 Fluid Rendering

Rendering a fluid requires a few preprocessing steps to merge nearby particles, creating a semi-smooth surface for which we can compute normals and, subsequently, refraction and reflection vectors. One key limitation of this approach is that we only compute surfaces *visible* from the camera, which means that information about occluded surfaces is discarded. This implies that

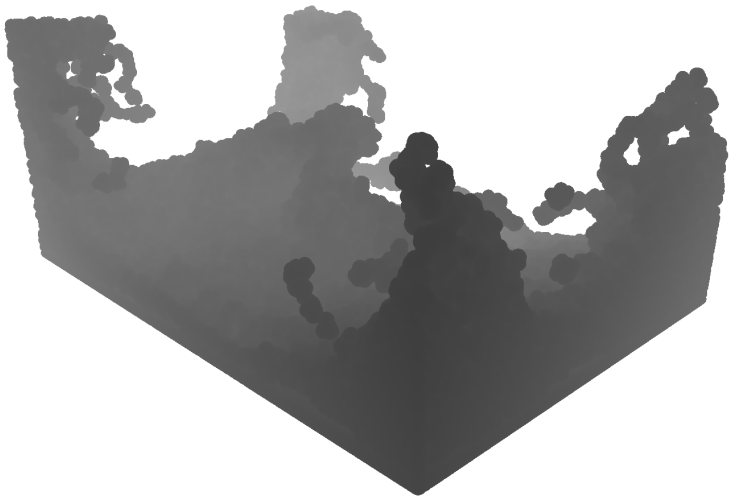
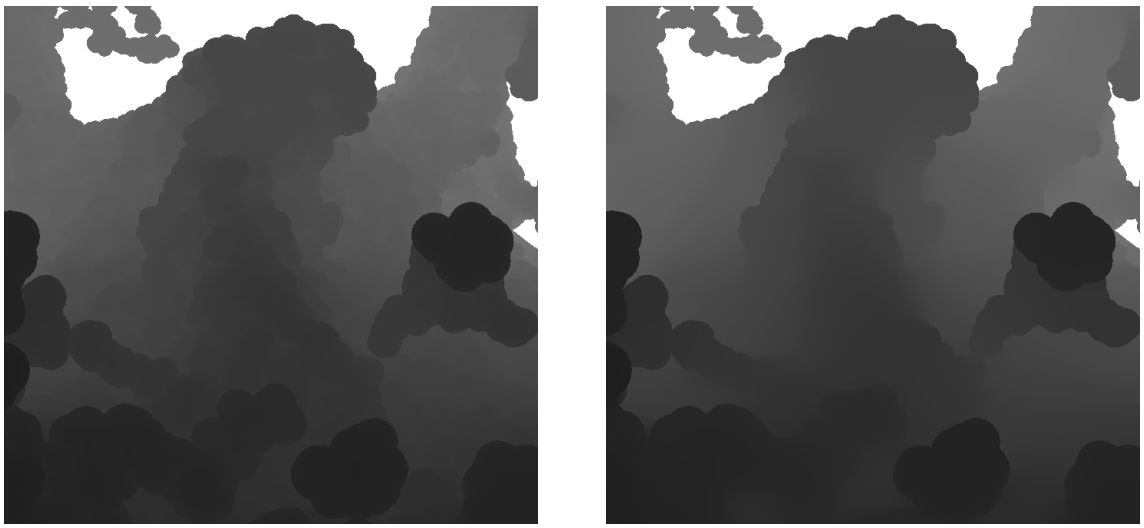


Figure 3: Per-particle depth map.

only the first refraction/reflection vector is computed, leading to visually plausible, albeit physically inaccurate, light distortions.
The first step consists of computing and storing a linear particle depthmap in a texture, as shown in Fig.3. This depthmap is used to filter out all those fragments not directly visible from the camera and, in later stages, becomes essential for reconstructing world-space coordinates from clip-space ones.
To render the depthmap, we used a bigger particle radius, in order to have a greater overlap between particles, making it easier to smooth them. To compute the correct sphere surface depth, we used the same approach used in Eq.1.



(a) Original depth map **(b)** Blurred depth map

Figure 4: Closeup of edge-preserving blurring applied to the depth map.

In order to fuse together particles close to each other we used a two-pass bilateral filter, which is a type of edge-preserving smoothing filter, meaning that if there is an abrupt depth variation in the image (i.e. corresponding to an object's edge) it does not blur it, as shown in Fig. 4.
The filter works as follows:

- **Vertical Pass:** the blurring only occurs on vertically adjacent texels and the resulting image is saved to a texture.
- **Horizontal Pass:** we blur horizontally adjacent texels, read from the vertical pass texture.

The bilateral filter performs edge-preserving smoothing by computing each output pixel as a weighted average of its neighbors, where the weights are determined by both spatial proximity and photometric similarity. Specifically, pixels that are closer in spatial distance and have similar intensity values contribute more significantly, while those with large intensity differences have a reduced influence. Formally, the formula for the 1D bilateral filter is:

$$w(i,j) = \exp\left(-\frac{(i-j)^2}{2\sigma_d^2} - \frac{(p_i - p_j)^2}{2\sigma_r^2}\right) \tag{3}$$

$$\hat{p_i} = \frac{\sum_{j\in\Omega_i} p_j \cdot w(i,j)}{\sum_{j\in\Omega_i} w(i,j)} \tag{4}$$

Here, $w(i,j)$ represents the weight assigned to the contribution of pixel j when computing the final intensity of pixel i. In other words, $w(\cdot)$ is a kernel function – specifically, a Gaussian kernel. The first term in Eq.3 accounts for the spatial distance between pixels i and j, while the second term weighs their intensity difference. The parameters σ_d and σ_r control the standard deviations of the spatial and range Gaussian kernels, respectively, determining their sensitivity to distance and intensity variations. In the implementation, these two parameters are named respectively `blurScale` and `blurFalloff`.
Finally, Eq.4 defines the complete blurring operation for pixel p_i, where the output is computed as a weighted sum of the intensities of all neighboring pixels j, within the local window Ω_i.

Although computationally efficient, the separation of the horizontal and vertical blur passes introduces a few artifacts near the edges (which are clearly visible in the normal map in Fig.5) but, in practice, are barely noticeable when the fluid is in motion.

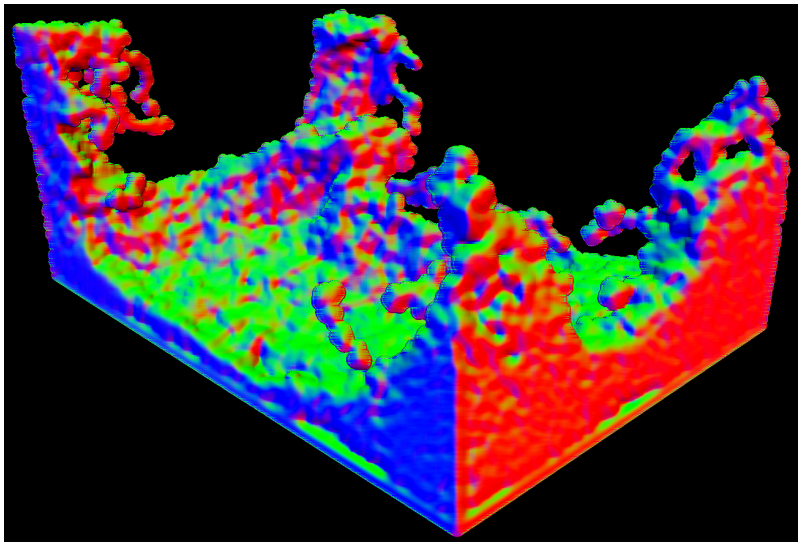


Figure 5: Normal map computed on the blurred depth map.

Normal reconstruction is performed by computing the cross product of two orthogonal displacement vectors, computed with respect to a UV texture coordinate, as shown in the following algorithm:

Algorithm 1 Normal computation from depthmap

function COMPUTENORMAL($UV, depth$)
  $c \leftarrow$ `uvToWorld`$(UV,\ depth)$
  $offset \leftarrow$ `texelSize`$(depthMap)$
  $d_x \leftarrow$ `uvToWorld`$(UV + (offset^{\{x\}},\ 0),\ depth)$
  $d_y \leftarrow$ `uvToWorld`$(UV + (0,\ offset^{\{y\}}),\ depth)$
  $N \leftarrow (d_x - c) \times (d_y - c)$ ▷ Cross product of displacement vectors
  return `normalize`(N)
end function

The `uvToWorld(·)` function converts a UV coordinate into a world one, by performing raycasting from the projection plane to the world, using the inverse view and projection matrices, as well as the depthmap for reconstructing the distance from the camera:

Algorithm 2 World coordinate from UV coordinate

function UVTOWORLD(UV, $depth$)
  $NDC \leftarrow (UV \cdot 2) - 1$ ▷ UV to NDC
  $V_c \leftarrow \begin{pmatrix} NDC^{\{x\}} & NDC^{\{y\}} & 1 & 1 \end{pmatrix}$ ▷ view vector in clip-space
  $V_v \leftarrow M^{-1}_{projection} \cdot V_c$ ▷ view vector in view-space
  $V_v^{\{z\}} \leftarrow -1$ ▷ forward direction in view-space is z=-1
  $V_w \leftarrow M^{-1}_{view} \cdot V_v$ ▷ view vector in world-space
  $d \leftarrow depth \cdot farPlane$ ▷ [0,1] linear depth to world-space depth
  return $cameraPosition + V_w \cdot d$ ▷ world coordinate of fluid surface
end function

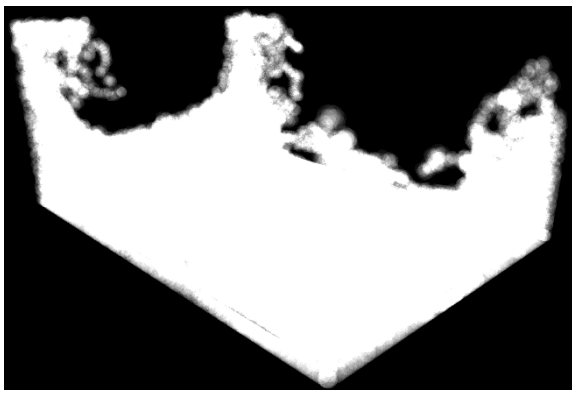
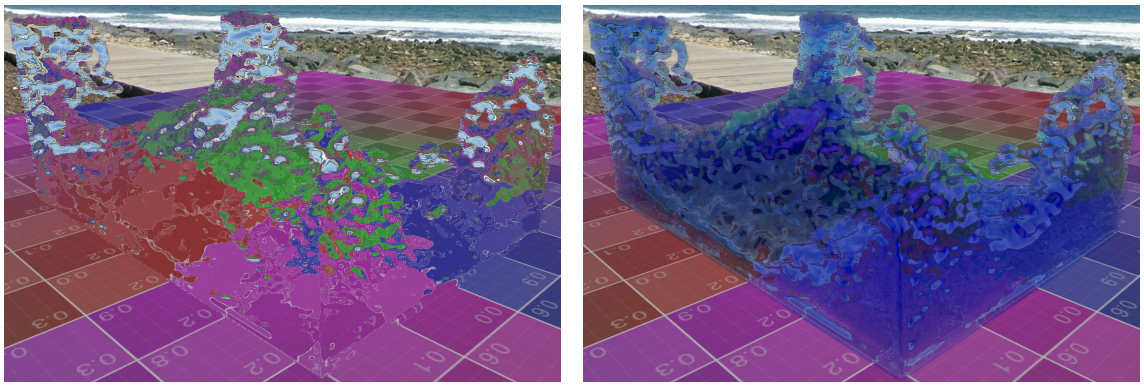


Figure 6: The particle thickness map.

Finally, to control the fluid's opacity and color, we compute its thickness and store the result in a texture. This is done by rendering all particles with OpenGL's additive blending enabled, so that the accumulated values in the texture represent fluid thickness, where 1 indicates maximum thickness and 0 indicates no fluid, as shown in Fig.6. Depending on various factors, such as number of particles and screen resolution, this step can get very resource-intensive, therefore, in our implementation it is performed at half the resolution.

3.1 Scene rendering and final composition



(a) Refractions only **(b)** Refractions + reflections + color

Figure 7: Final fluid rendering, with refractions only (a) and with reflections and light extinction coefficients set (b).

The scene used in this project consists of a simple plane on which the fluid is located, and as a background, we used a generic cubemap[1]. The plane rendering is saved to a texture; this way when we compute refraction and reflection vectors we can easily check whether a ray-plane intersection occurs and sample either the plane texture or the cubemap accordingly. The main limitation of this approach is that, in case of more complex scenes, we need to update accordingly the ray-intersection tests, as well as the texture sampling logic.

The final rendering steps can be summarized as follows:

- **Refracted light**: after computing the refraction vector, sample the background texture (either the plane or the cubemap). Using the thickness texture and the color extinction coefficients of the fluid, we can compute its transmittance as follows:

$$T = \exp\left(fluidColor \cdot thickness\right) \tag{5}$$
$$C = C.rgb \odot T \tag{6}$$

  Here T is the vector containing the transmittance coefficients for each color channel, and C is the final color of the refraction ray.

- **Reflected light**: compute the reflection vector, sample the appropriate background texture.

- **Combine reflections and refractions**: we use the Schlick approximation [4] of the Fresnel reflectance equations to compute the reflection coefficient α, and the resulting color is a linear interpolation of the reflected and refracted color:

$$F_0 = \frac{(R_{air} - R_{water})^2}{(R_{air} + R_{water})^2} \tag{7}$$
$$\alpha = F_0 + (1 - F_0) \cdot (1 - \cos\theta)^5 \tag{8}$$
$$C = \alpha \cdot C_{refraction} + (1 - \alpha) \cdot C_{reflection} \tag{9}$$

  Where R_i is the Index of Refraction of a given material i, θ is the angle between the surface normal and the half vector, $C_{\{...\}}$ is the sampled background color, coming either from a reflection or a refraction.

By experimenting with the color extinction coefficients of the fluid, we can get the results shown in Fig.7b and Fig.8.

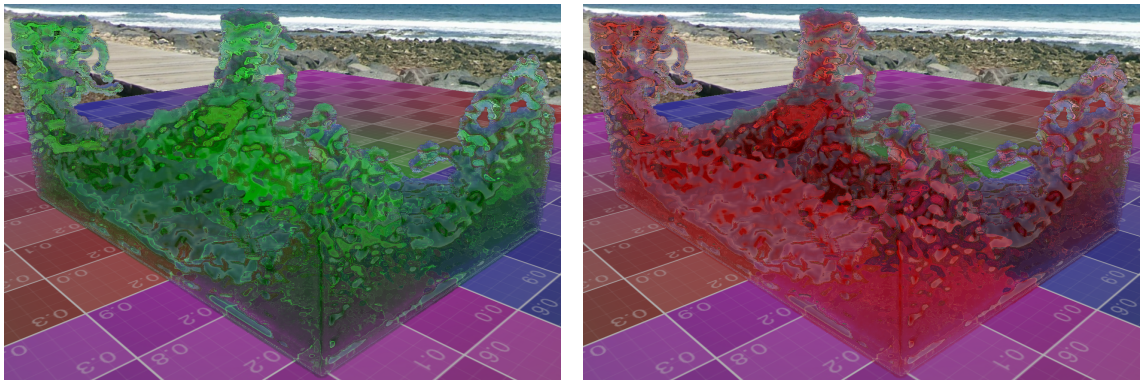


Figure 8: Example of various fluid colors.

[1] https://www.humus.name/index.php?page=Cubemap&item=Tenerife2

References

[1] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Seminal graphics: pioneering efforts that shaped the field*. ACM, 1998.

[2] Wladimir J van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proc. I3D*, 2009.

[3] Simon Green. Screen space fluid rendering for games. In *GDC*, 2010.

[4] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*. Wiley Online Library, 1994.