

Gaussian Splashing: Unified Particles for Versatile Motion Synthesis and Rendering

Supplementary Material

7. Simulation details

7.1. Position-based dynamics

PBD/XPBD treats a dynamic system as a set of N vertices, i.e., $\mathbf{x} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_N]^\top$ and M constraints, i.e., $\mathbf{C}(\mathbf{x}) = [C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_M(\mathbf{x})]^\top$. Here, \mathbf{x} represents the position of vertices and $\mathbf{C}(\mathbf{x})$ represents the set of constraints. Specifically, the total system potential U is defined as a quadratic form of all the constraints such that $U = \frac{1}{2} \mathbf{C}^\top(\mathbf{x}) \boldsymbol{\alpha}^{-1} \mathbf{C}(\mathbf{x})$. Here, $\boldsymbol{\alpha}$ is the compliance matrix, i.e., the inverse of the constraint stiffness. For example, if there are only two vertices and they form a mass-spring system, constraint and compliance matrix could be written as $\mathbf{C}(\mathbf{x}) = [\|\mathbf{x}_0 - \mathbf{x}_1\| - d_0]^\top$ and $\boldsymbol{\alpha} = [k]$, where d_0 is the rest length of the spring and k is the stiffness of the spring.

Motion at each time step can be solved by minimizing the system energy. However, PBD/XPBD offers an easy and efficient simulation modality, converting the variational optimization to the so-called constraint projections.

XPBD estimates an update of constraint force (i.e., the multiplier) $\Delta\lambda$ by solving:

$$[\Delta t^2 \nabla \mathbf{C}(\mathbf{x}) \mathbf{M}^{-1} \nabla \mathbf{C}^\top(\mathbf{x}) + \boldsymbol{\alpha}] \Delta\lambda = -\Delta t^2 \mathbf{C}(\mathbf{x}) - \boldsymbol{\alpha} \lambda, \quad (13)$$

where Δt is the time step size, and \mathbf{M} is the lumped mass matrix. The update of the primal variable $\Delta\mathbf{x}$ can then be computed as:

$$\Delta\mathbf{x} = \mathbf{M}^{-1} \nabla \mathbf{C}^\top(\mathbf{x}) \Delta\lambda. \quad (14)$$

The parallelization of XPBD is enabled with a Gauss-Seidel-like scheme, which computes $\Delta\lambda_j$ at each constraint C_j independently:

$$\Delta\lambda_j \leftarrow \frac{-\Delta t^2 C_j(\mathbf{x}) - \alpha_j}{\Delta t^2 \nabla C_j \mathbf{M}^{-1} \nabla C_j^\top + \alpha_j}. \quad (15)$$

A typical XPBD simulation loop is shown in Algorithm 1.

7.2. Position-based fluids

We employ the Position-Based Fluids (PBF) [40] as our Lagrangian fluid synthesizer. PBF is based on PBD, which means it also use constraint projections to simulate fluid behaviour. In PBF, fluid is composed of a large amount of particles. To enforce the fluid incompressibility, PBF imposes a density constraint C_i^ρ on each particle, maintaining the integrated density ρ_i computed by the SPH kernel as:

$$C_i^\rho = \frac{\rho_i}{\rho_0} - 1 = \sum_j \frac{m_j}{\rho_0} W(\mathbf{p}_i - \mathbf{p}_j, r) - 1, \quad (16)$$

Algorithm 1 XPBD simulation loop for time step $n + 1$

```

1: predict position  $\hat{\mathbf{x}} \leftarrow \mathbf{x}^n + \Delta t \mathbf{v}^n + \Delta t^2 \mathbf{M}^{-1} \mathbf{f}_{ext}(\mathbf{x})^n$ 
2:
3: initialize solve  $\mathbf{x}_0 \leftarrow \mathbf{x}$ 
4: initialize multipliers  $\lambda_0 \leftarrow \mathbf{0}$ 
5: while  $i < \text{solverIterations}$  do
6:   for all constraints do
7:     compute  $\Delta\lambda$  using Eq. 15
8:     compute  $\Delta\mathbf{x}$  using Eq. 14
9:     update  $\lambda_{i+1} \leftarrow \lambda_i + \Delta\lambda$ 
10:    update  $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \Delta\mathbf{x}$ 
11:   end for
12: end while
13: update positions  $\mathbf{x}^{n+1} \leftarrow \mathbf{x}_i$ 
14: update velocities  $\mathbf{v}^{n+1} \leftarrow \frac{1}{\Delta t} (\mathbf{x}^{n+1} - \mathbf{x}^n)$ 

```

where m_j is the mass of particle j . \mathbf{p}_i is the position of particle i , W is the SPH kernel function and r is the kernel radius. Intuitively, projecting this constraint to 0 ensures that the density at the current time remains consistent with the initial state. We use the following cubic SPH kernel:

$$W(\mathbf{p}, r) = \begin{cases} \frac{8}{\pi r^3} (6q^2(q-1) + 1), & 0 \leq q \leq 0.5 \\ \frac{16}{\pi r^3} (1-q)^3, & 0.5 < q \leq 1 \\ 0, & \text{otherwise} \end{cases} \quad (17)$$

where $q = \frac{\|\mathbf{p}\|}{r}$. The Jacobian of constraint is computed as:

$$\nabla_{\mathbf{p}_k} C_i^\rho = \begin{cases} \sum_j \frac{m_j}{\rho_0} \nabla_{\mathbf{p}_i} W(\mathbf{p}_i - \mathbf{p}_j, r), & k = i \\ \frac{m_j}{\rho_0} \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_j, r), & k = j. \end{cases} \quad (18)$$

The gradient of the kernel function is:

$$\nabla_{\mathbf{p}} W(\mathbf{p}, r) = \begin{cases} \frac{48}{\pi r^5} (3q-2) \mathbf{p}, & 0 \leq \frac{\|\mathbf{p}\|}{r} \leq 0.5 \\ -\frac{48}{\pi r^5} \frac{(1-q)^2}{q} \mathbf{p}, & 0.5 < \frac{\|\mathbf{p}\|}{r} \leq 1 \\ 0, & \text{otherwise} \end{cases} \quad (19)$$

GSP also includes a position-based surface tension model [73] to better capture the dynamics of the fluid surface. We first detect whether a particle (i.e., a Gaussian kernel) is on the fluid surface based on occlusion estimation. Specifically, we encapsulate a particle with a spherical cover or screen. Each of its neighboring particles generates a projection on the screen (because a particle has a finite

volume). The particle is considered on the fluid surface if the total projection area from its neighbors is below a given threshold.

In the original paper [73], surface detection is implemented on the CPU. It is noteworthy that surface detection can be parallelized on the GPU to expedite the simulation, as the calculation of each particle’s occluding ratio on the screen is independent of the others.

For each neighboring particle, its occluding area on the spherical screen is calculated as follows:

$$\begin{aligned}\theta &= \tan^{-1}\left(\frac{\Delta p_y}{\Delta p_x + \Delta p_z^2}\right) \\ \phi &= \tan^{-1}\left(\frac{\Delta p_x}{\Delta p_z}\right) \quad (20) \\ \Delta\theta &= \tan^{-1}\frac{R}{\|\Delta p\|^2 - R^2} \\ \Delta\phi &= \Delta\theta\end{aligned}$$

where Δp is the vector from the detection particle to the neighboring particle and R is the particle radius. The shadowed area on the spherical screen is then $[\theta - \Delta\theta, \theta + \Delta\theta] \times [\phi - \Delta\phi, \phi + \Delta\phi]$. We parameterize the screen as an 18×36 environment map, with each column of the environment map corresponding to 18 bits of an integer. We then mask 36 integers and count the mask ratio.

After detecting surface particles on the fluid, we apply tension on the surface. Tensions tends to minimize surface area. Therefore, PBF applies an area constraint to each surface particle to minimize the local surface area nearby. We start by calculating the normal n_i of surface particles i as:

$$n_i = \text{normalize}(-\nabla_{p_i} C_i^\rho), \quad (21)$$

where $C_i^\rho = 0$ indicates the particle is inside the fluid, and $C_i^\rho = -1$ indicates it is outside. After that, we project the neighboring surface particles onto a plane perpendicular to n_i and triangularize the plane. The area constraint can then be built as:

$$C_i^A = \sum_{t \in T(i)} \frac{1}{2} \|(\mathbf{p}_{t^2} - \mathbf{p}_{t^1}) \times (\mathbf{p}_{t^3} - \mathbf{p}_{t^1})\| \quad (22)$$

where $T(i)$ is the set of neighboring triangles for particle i . We use the 2D Delaunay triangulation to construct the triangles on the local surface. This process is sequential and cannot be parallelized on the GPU. However, it is sufficiently fast and we translate it from CPU to GPU.

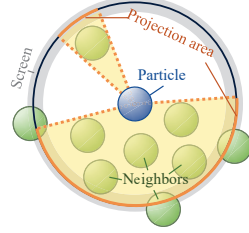


Figure 12. **Detection of surface particles.** An interior particle is detected if its screen is widely shadowed by its neighbors. A boundary particle is detected if at least one part of the particle’s screen is not shadowed.

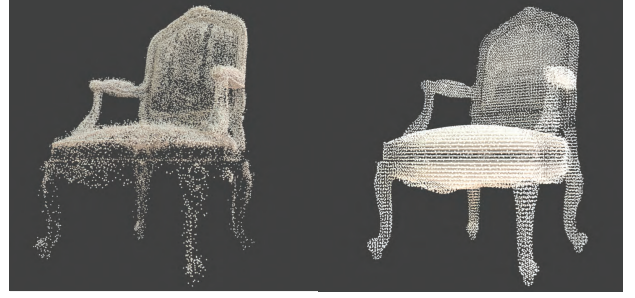


Figure 13. **Different sampling strategies.** We compare the results of different sampling strategies: (left) fill the particle based on the density grid calculated using Gaussian kernels [72], and (right) uniformly sample within NeuS reconstruction. The point distribution generated by vanilla 3DGS is uneven, which hardly samples the legs or seat of the chair.

To promote a more uniform particle distribution, additional distance constraints are introduced to push apart particles that are too close to each other:

$$C_{ij}^D = \min\{0, \|\mathbf{p}_i - \mathbf{p}_j\| - d_0\}, \quad (23)$$

where d_0 is the distance threshold. The Jacobian of aboved constraints are:

$$\begin{aligned}\nabla_{t^1} C_t^A(\mathbf{p}) &= \frac{(\mathbf{p}_{t^2} - \mathbf{p}_{t^1}) \times (\mathbf{p}_{t^3} - \mathbf{p}_{t^1}) \times (\mathbf{p}_{t^3} - \mathbf{p}_{t^2})}{2 \|(\mathbf{p}_{t^2} - \mathbf{p}_{t^1}) \times (\mathbf{p}_{t^3} - \mathbf{p}_{t^1})\|}, \\ \nabla_{t^2} C_t^A(\mathbf{p}) &= \frac{(\mathbf{p}_{t^3} - \mathbf{p}_{t^2}) \times (\mathbf{p}_{t^1} - \mathbf{p}_{t^2}) \times (\mathbf{p}_{t^1} - \mathbf{p}_{t^3})}{2 \|(\mathbf{p}_{t^3} - \mathbf{p}_{t^2}) \times (\mathbf{p}_{t^1} - \mathbf{p}_{t^2})\|}, \\ \nabla_{t^3} C_t^A(\mathbf{p}) &= \frac{(\mathbf{p}_{t^1} - \mathbf{p}_{t^3}) \times (\mathbf{p}_{t^2} - \mathbf{p}_{t^3}) \times (\mathbf{p}_{t^2} - \mathbf{p}_{t^1})}{2 \|(\mathbf{p}_{t^1} - \mathbf{p}_{t^3}) \times (\mathbf{p}_{t^2} - \mathbf{p}_{t^3})\|}, \\ \nabla_i C_{ij}^D(\mathbf{p}) &= \begin{cases} 0, & \|\mathbf{p}_i - \mathbf{p}_j\| > d_0, \\ \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|}, & \text{Others.} \end{cases} \\ \nabla_j C_{ij}^D(\mathbf{p}) &= \begin{cases} 0, & \|\mathbf{p}_i - \mathbf{p}_j\| > d_0, \\ \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_i - \mathbf{p}_j\|}, & \text{Others.} \end{cases} \end{aligned} \quad (24)$$

8. Sampling and interpolation details

In practice, we found that an uneven distribution of Gaussian kernels results in unstable and inaccurate motion synthesis, while a distribution that is too uniform can detrimentally affect rendering quality. Gaussian kernels tend to distribute primarily unevenly around the surfaces and edges of objects, leading to inaccurate boundary descriptions, which are crucial for interactions between objects in simulation. Conversely, adaptively distributed anisotropic Gaussian kernels are key to representing the spatially varying texture on the object. To address this issue, we maintain two separate sets of points: one sampled from the NeuS mesh surface for simulation, and the other consisting of

trained Gaussian kernels for rendering. We compare the results of directly sampling from trained Gaussian kernels to those of sampling from NeuS in Fig. 13. The former method can result in sparsely sampled regions, especially for objects with thin parts, potentially affecting simulation quality.

We then discuss how to animate trained Gaussian kernels for rendering dynamics. Denote the set of trained Gaussian kernels for rendering as S_r , and the set of sampled points from NeuS for simulation as S_s . At time 0, we initialize GMLS kernels on S_s and find the k nearest neighbors $\{\mathbf{p}_{s,j}^0 : j \in \mathcal{N}(i)\}$ from S_s for each point $\mathbf{p}_{r,i}$ in S_r . Here, $\mathcal{N}(i)$ denotes the set of k nearest neighboring particles' indices of $\mathbf{p}_{r,i}$ in S_s at time 0. As the simulation or motion synthesis proceeds, the position $\mathbf{p}_{s,j}^n$ evolves with time step n . We then update $\mathbf{p}_{r,j}$ by interpolating from $\{\mathbf{p}_{s,j}^n : j \in \mathcal{N}(i)\}$ with the pre-built GMLS kernel. The interpolation of deformation is achieved in the same way, replacing the physical quantity position \mathbf{p} to deformation gradient F .

9. Rendering details

Shadow As shadows are crucial to the visual outcomes in dynamic scenes, we re-engineer nearly-soft shadows [13] into our system to enhance realism. Following shadow mapping, we splat all Gaussian kernels to a camera positioned at the point light's location, which we denote as light-view. The point light is aligned with the direction of the significantly bright light in the environment map. The resolution of the light-view image is three times that of the original image resolution to address visual discrepancies caused by under-sampling.

We then reproject the points seen in the camera view to the light-view and compare their depths to the previously splatted light-view depth image. A larger depth indicates that the point is occluded by a nearer point and will therefore cast a shadow. A more robust variance method is discussed in [13]. Softer shadows can be achieved by blurring and averaging the light-view depth image. We compute the shadowing probability using Chebyshev's inequality [13] and store the results in a shadow map. Finally, we composite the rendered image with the shadow map to achieve nearly-soft dynamic shadows.

Spray, foam, and bubble To enhance the realism of fluids, foam, spray and bubble particles are synthesized with [22] as a post-processing step. Fluid-air mixtures are generated at the crest of the wave and in the impact zone of the wave. Spray, foam, and bubble particles are advected by the fluid and dissipate within their predefined lifetimes.

We splat these particles into a foam intensity image using modified additive splatting. Different types of particles use different kernels during splatting [1]. We preferred a larger

overall intensity for surface foam particles to increase their visibility, while we used a comparatively smaller intensity value for spray particles to make them less prominent. Furthermore, we used hollow circle structures for the bubble particles to make their appearance more convincing underwater. The kernel typically has a radius of 2 pixels. However, in practice, we found that the kernel radius should be scaled based on the particle's depth in the view, as particles near the camera occupy more of the view compared to those farther away. Finally, we apply a curve [1] on the foam intensity image to scale it into $[0, 1]$ and compose it with rendering.

10. Implementation details

We set the simulation time step as 0.005 seconds throughout the simulations. In our PBD solver, we used 10 iterations for fluids and 50 iterations for solids for our experiments, since mass particles on the solid models are more strongly coupled than the ones in the fluid. During the PBF simulation, the surface particles of fluids are updated every two time steps.

11. Comparison with PhysGaussian

PhysGaussian [72] is a recent work that also integrates physical models into 3DGS. However, its rendering module does not support dynamic refraction and reflection, both of which are essential for fluid rendering. PhysGaussian employs spherical harmonics for shading, complicating effective manual editing and preventing the accurate rendering of dynamic fluids. In Fig. 14, we present a side-by-side comparison. In this experiment, we transform the hot dog on the plate to water and allow it to interact with the stationary plate. Our approach clearly demonstrates improved rendering quality for fluids.

12. More results

We show the results of **Chair** (Fig. 15), **Waves** (Fig. 8), **Garden** (Fig. 16), **Lego** (Fig. 17), **Cup & dog** (Fig. 18), **Headset** (Fig. 19), **Can** (Fig. 20), **Astronaut** (Fig. 23), **Ficus** (Fig. 21), and **Bulldozers** (Fig. 22). For better visualization, please refer to the supplementary video.



Figure 14. **Comparison with PhysGaussian [72].** We compare our method with PhysGaussian [72] for fluid editing and rendering. The first row presents our results, while the second row shows those of PhysGaussian. In this experiment, we transform a hot dog placed on a plate into fluid, allowing it to interact dynamically with the stationary plate. PhysGaussian struggles to render realistic fluid effects, whereas our method effectively renders fluid and accurately captures water highlights, yielding significantly more realistic results.

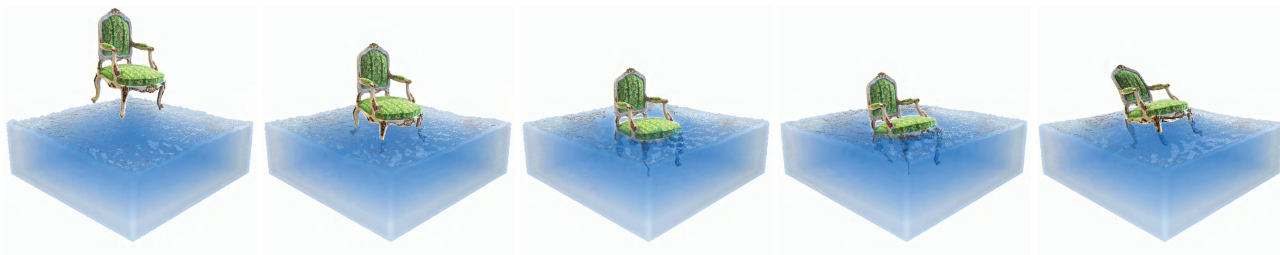


Figure 15. **Chair.** A soft chair falls into the pool, causing deformation and ripples.

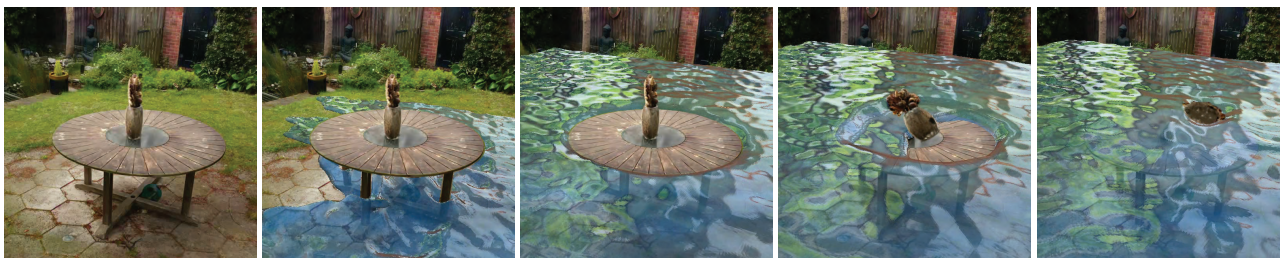


Figure 16. **Flooding garden.** Water leaks into the garden and submerge the table. As the water level goes up, the surface gets more vibrant and washes the potted plant away.

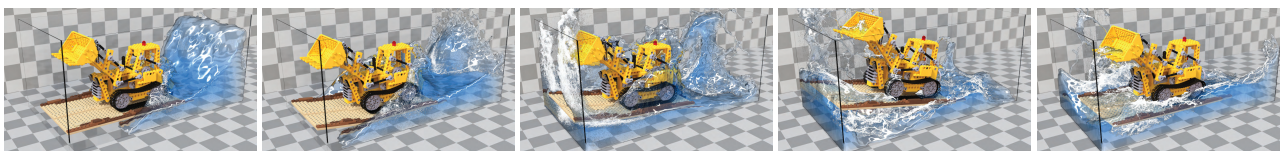


Figure 17. **Splashing LEGO.** Through the two-way coupling dynamics, the LEGO bulldozer is animated to surf on the splashing waves.

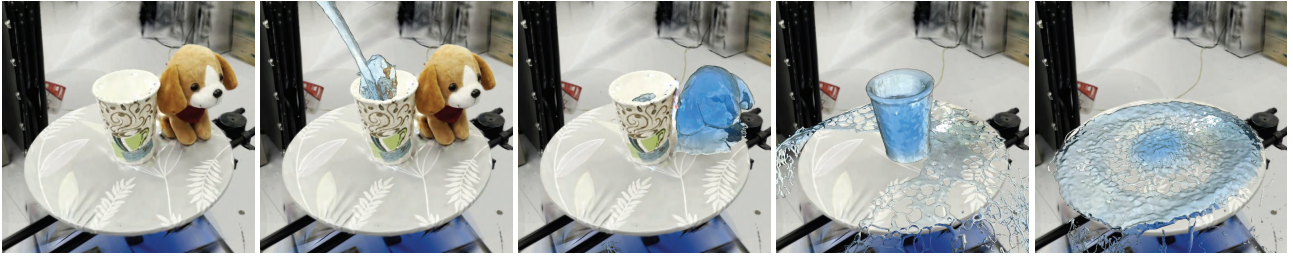


Figure 18. **“Everything is water”**. Pouring water into the paper cup on the table and transforming the cup and the dog toy into water. The water spills out.



Figure 19. **Headset waterfall**. Water flows from a headset hanging above an office desk, resembling a faucet. Due to surface tension, the water forms droplets as it falls, sliding down the computer screen and splashing onto the desk, creating a puddle.



Figure 20. **Water droplets on can**. Droplets of water fall onto the surface of a soda can, coalesce due to surface tension and gradually overflow.



Figure 21. **Deformable ficus**. A deformable ficus plant undergoes continuous shape changes as it is dragged and manipulated by external forces.

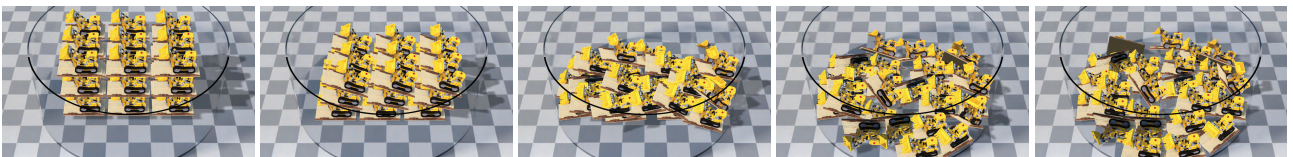


Figure 22. **LEGO bulldozers in glass bowl**. A collection of LEGO bulldozer rigid bodies fall into a round glass box, colliding with each other. They cast shadow on the ground and eventually stack and scatter throughout the box.

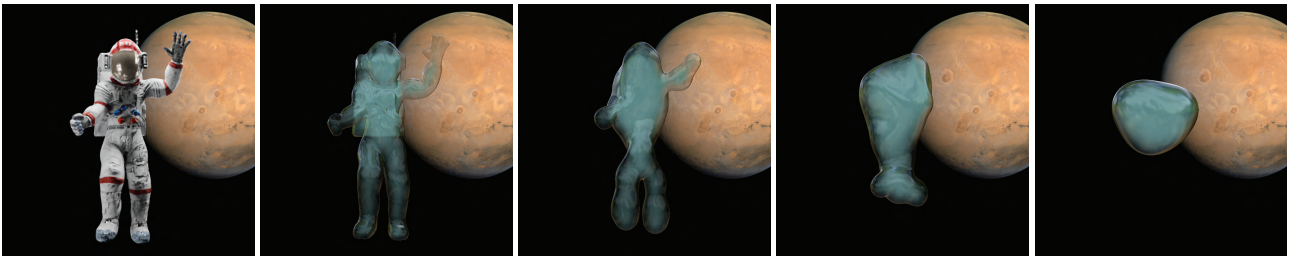


Figure 23. **Black magic.** An astronaut in space struck by the black magic of the Trisolarans, and get transformed into a water sphere.