CS 488 Project Proposal

# Bottles & Water

*Palaksha Drolia, Awab Qureshi*

*pdrolia, a9quresh*

# Final Project

## *Topics/Purposes*

- Physics-based fluid simulation
- Physics-based rigid body simulation that interacts cohesively with the fluid

## *Statement*

This project focuses mainly on simulating the interaction between a bottle and a water body based on realistic physics. The bottle undergoes rigid body dynamics under the effect of gravity and collision. Meanwhile, the water is rendered using a 2D shallow-water model with realistic wave generation and interactions involving buoyancy and drag. The entire scene is rendered using rasterisation via OpenGL.

The project begins by setting up a basic OpenGL pipeline for rasterisation. Next, we will need to implement a 2D shallow-water physics model that will dynamically render waves in response to external forces while accounting for displacement. This also involves rasterising the water surface with correct transparency adjustments. Reflection may also be implemented via multi-pass rendering, Screen-Space Reflections, or Cube mapping to improve the realism of the water surface. We will also implement rigid-body physics for the bottle and its shards via Verlet integration. The bottle shattering is done by specifying predefined fragments for the bottle. The interaction between the bottle and the water body would also account for the buoyancy and drag equations. These effects must be combined to form a cohesive scene.

The project is interesting because it involves multiple different physics-based components that need to interact with each other. The challenge lies in creating a realistic fluid simulation, especially when simulating its interactions with the bottle. Rasterizing the water surface convincingly is also an interesting problem to tackle. Additionally, since this project will be done in OpenGL, we will also need to familiarize ourselves with its API, which adds another layer of complexity.

Through this project, we will learn to simulate fluids and rigid bodies with physics. We will also understand how to model the interactions between them in a physically plausible manner. The use of OpenGL as the rendering engine will also improve our understanding of the graphics pipeline. We will also grasp the intricacies of rasterisation and how we can approximate more complex effects that were beyond the assignment content.

## *Technical Outline*

We plan on using a small and basic subset of OpenGL for rasterisation to take some load off the CPU. Drawing triangles to the screen will be done by the standard process described in [1]. We shall similarly integrate the transformation matrix and our shade function into the OpenGL pipeline, and they shall form our vertex and fragment shader, respectively. Our focus is not to harness all OpenGL features to the best of our ability, instead it is to simply replace CPU rasterisation with a GPU one.

### 1.3.1 Shallow-Water Simulation

The main focus of our project is the physics-based simulation of a shallow, large body of water: the pool. In order to maintain a level of interactability, we shall, instead of a full three-dimensional fluid simulation, reduce our problem to a two-dimensional height field.

For this, we employ the shallow water equations as given in [2],

$$\frac{Dh}{Dt} = -h(\nabla \cdot v)$$
$$\frac{Dv}{Dt} = -g\nabla\eta + a^{ext}$$

where $h$ is the depth of the water, $H$ is the $y$-coordinate of the terrain on the bottom, $\eta = H + h$ is the $y$-coordinate of the water's surface, $v$ is the vector $(u, w)$ representing the horizontal velocity of the fluid, $g$ is gravity, $a^{ext}$ is the external acceleration, and $D$ is the material derivative operator as given in the course slides [3],

$$\frac{D}{Dt} := \frac{\partial}{\partial t} + (v \cdot \nabla)$$

For better accuracy, we implement a staggered grid for our discretised simulation as specified briefly in the course slides [3] and [2]. We store the heights of a cell, $h_{i,j}$ and $H_{i,j}$ at its centre. And we store the velocity components $u_{i+\frac{1}{2},j}, w_{i,j+\frac{1}{2}}$ on the faces. When computing values not stored, we Bilinearly interpolate between values.

The grid is first updated with values generated by computing the advection of height and velocity. We use the semi-lagrangian method as proposed in [3] in order to solve the advection of $h_{i,j}$, $u_{i+\frac{1}{2},j}$ and $v_{i,j+\frac{1}{2}}$. Let $x_u = ((i + \frac{1}{2})\Delta x, j\Delta x)$ be the position of the grid cell at $u_{i+\frac{1}{2},j}$ and similarly define $x_v$ and $x_h$ We thus compute the new values,

$$u^{n+1}_{i+\frac{1}{2},j} = interp(x_u - \Delta t \cdot (u_{i+\frac{1}{2},j}, w_{i,j})^T)$$
$$w^{n+1}_{i,j+\frac{1}{2}} = interp(x_v - \Delta t \cdot (u_{i,j}, w_{i,j+\frac{1}{2}})^T)$$
$$h^{n+1}_{i,j} = x_h - \Delta t \cdot (i\Delta x, j\Delta x)$$

where we interpolate to find values that we do not store.

The heights are integrated by adding the following,

$$h_{i,j} = -\left(\frac{\overline{h}_{i+\frac{1}{2},j}u_{i+\frac{1}{2},j} - \overline{h}_{i-\frac{1}{2},j}u_{i-\frac{1}{2},j}}{\Delta x} + \frac{\overline{h}_{i,j+\frac{1}{2}}w_{i,j+\frac{1}{2}} - \overline{h}_{i,j-\frac{1}{2}}w_{i,j-\frac{1}{2}}}{\Delta x}\right)\Delta t$$

as suggested by [2], and [3]. Here we implement a heuristic provided by [2], where instead of linearly interpolating to find the values of $\overline{h}_{\_,\_}$, we instead evaluate it to be equal to $h$ in the upwind direction.

The velocities are updated, as [2] suggests, taking the gradient of the water height. For our staggered

velocities, we add the following,

$$u_{i+\frac{1}{2},j} \mathrel{+}= \left(\frac{-g}{\Delta x}(\eta_{i+1,j} - \eta_{i,j}) + a_x^{ext}\right)\Delta t$$
$$w_{i,j+\frac{1}{2}} \mathrel{+}= \left(\frac{-g}{\Delta x}(\eta_{i,j+1} - \eta_{i,j}) + a_z^{ext}\right)\Delta t$$

where $a^{ext}$ is the external acceleration.

For boundary conditions, our pool shall either have a well-defined boundary that must reflect waves, or we shall simulate an open water scene with seemingly no boundary. For the former, we shall carry out the method suggested in [3] and at the boundary set the heights to the same as their neighbours, and set the velocity component into the wall to be zero. For the latter, we carry out the method suggested by [2] and make the boundary absorb the waves.

### 1.3.2 Rasterising Water Surface

We rasterise the water surface with the method described in [2]. The height field of our fluid is already a gird of quads, with each $(i, j)$ vertex having height $\eta_{i,j}$ that we may split into triangles. We copy over the vertex data to the GPU and split the quads to emit triangles in a geometry shader.

Additionally, as suggested by [2], we slice the quad into triangles across the following diagonal,

$$\begin{cases} (i,j) - (i+1, j+1) & \text{if } \eta_{i,j} + \eta_{i+1,j+1} > \eta_{i+1,j} + \eta_{i,j+1} \\ (i+1,j) - (i, j+1) & \text{else} \end{cases}$$

since picking the diagonal that aligns with the wave's crests reduces artefacts.

For transparency, we rely on OpenGL's blending. After enabling OpenGL's blending, we draw the opaque pool first and then draw the transparent water surface as suggested in [4].

### 1.3.3 Rigid-body Physics

For rigid-body physics, we employ Verlet integration as suggested by [5]. A force acts on the centre of mass of the rigid body whose next coordinate is computed via

$$x_{n+1} \approx x_n + (x_n - x_{n-1}) + \Delta t^2 \frac{F_n}{m}$$

where $x_n$ is the vector that represents the coordinates of the body, $m$ is its mass, $\Delta t$ is the time step, and $F_n$ is the force acting on it. For detecting collisions with objects other than the water's surface, we use a simple bounding box around the bottle. We plan to use force-based collisions since we do not have any complex interactions other than the water's surface. As suggested in [5], we accomplish this by checking if there is a collision, then computing the resultant velocity with a coefficient of restitution. We also keep track of a body's angular velocity so that it may spin about its axis.

### 1.3.4 Rigid-body and Water Interaction

When a rigid body lands in the pool, we wish to modify the height and velocity of the fluid. To accomplish this, we utilise Algorithm 2 as given in [2]. We essentially subdivide the rigid body's triangles into smaller triangles until their area falls below $\Delta x^2$ where $\Delta x$ is our grid spacing recursively. This subdivision allows fine-grained spatial resolution for capturing interactions accurately between the body and the fluid. For each of these small triangles, the centroid's position $p = (p_x, p_y, p_z)$ and its velocity $v = (v_x, v_y, v_z)$ is computed via barycentric interpolation. We use the triangle's normal $n$ to determine the direction and magnitude of fluid displacement.

The velocity's magnitude relative to the vertical direction is then used to determine how many substeps to divide the current simulation timestep into,

$$num\_substeps = \max\left\{1, \left\lfloor |v - v_y y| \frac{\Delta t}{\Delta x} + 0.5 \right\rfloor\right\}$$

This is because more substeps allow smoother and more stable application of forces along the triangle's trajectory. For each substep, then, we advance the centroid position along its velocity, identify the fluid grid cell closest to this position, $(i, j)$ and calculate the depth of the centroid relative to the fluid surface height. Upon finding the centroid submerged, we compute a decay factor that exponentially reduces influence with depth, reflecting that deeper submerged parts affect the fluid less. The fluid surface's height at the grid cell is then updated by adding a volume displacement proportional to the triangle's area, velocity, and decay factor,

$$h_{i,j} \mathrel{+}= e^{-\overbrace{(\eta_{i,j} - p_y)}^{depth}} \frac{\overbrace{n \cdot v_{rel} A \Delta t}^{V_{disp}}}{num\_substeps(\Delta x)^2}$$

where $v_{rel} = v - v_{fluid}$. The fluid surface's velocity is then updated at the corresponding staggered grid faces by pushing them towards the triangle centroid's velocity, scaled by a coefficient,

$$coeff = \min\left\{1, \frac{e^{-\overbrace{(\eta_{i,j} - p_y)}^{depth}}}{5} \frac{\overbrace{(\eta_{i,j} - p_y)}^{depth}}{\eta_{i,j}} sign \frac{\Delta t}{(\Delta x)^2} A\right\}$$

where $sign = \begin{cases} 1 & \eta_y > 0 \\ -1 & \text{else} \end{cases}$

The fluid must also interact with the rigid body. For this we consider buoyancy, drag and lift forces. We compute the sum of these forces $F_i = f_{buoyancy}, f_{drag}, f_{list}$ for each centroid of our subdivided triangle grid from before. The total force acting on the body at its centre of mass is given by $F = \sum F_i$. And each $F_i$ produces a torque about the body's centre of mass, so the total torque generated is $\tau = \sum r_i \times F_i$, where $r_i$ the displacement from $p$ to the centre of mass, which can be used to update the angular velocity of the body. We compute these forces by the very straightforward equations 14, 15, 16 and 17 as given in [2] which we do not copy here for brevity. The equations let us adjust coefficients $C_D$, $C_L$ and $\omega$ for the drag, lift and effective area respectively. We shall likely require some experimentation to settle on values for them.

#### 1.3.5 Bottle Shattering

Optionally, we aim to simulate the bottle shattering via a pre-defined mechanism. The plan is for it to "shatter" into, by which we mean replaced by, pre-defined large pieces. Each piece will be a separate rigid body that can then be simulated. When we detect a collision, we shall check if the force crossed a certain threshold. In that case this shatter event shall occur. When this happens, we shall also increase the water height in the nearby grid cells, if it happened near the water, thus simulating water increase in the pool.

## Objectives

1. Replace CPU rasterisation with OpenGL rasterisation pipeline. Write vertex and fragment shaders to implement MVP matrix, diffuse and specular texture maps, and Phong shading.
2. Model the objects required for the scene, such as the bottle.
3. Simulate 2D shallow water as a height field by integrating height and velocity as described in the technical outline.
4. Solve advection equations and add boundary conditions to simulate the waves accurately as described in the technical outline.
5. Implement a geometry shader to rasterise the surface of the water such that it appears continuous as described in the technical outline.
6. Implement rigid-body physics for the bottle using Verlet integration. Handle collisions with the pool and other objects with bounding boxes as described in the technical outline.
7. Simulate change in water levels and waves when objects collide as described in the technical outline.
8. Implement fluid-body interaction (buoyancy and drag) as described in the technical outline.

### *Extra Objectives*

Additionally, provided we have time to spare and things go well, we have the following extra objectives that we may implement. Most may not be included in the technical overview.

1. Simulate bottle shattering on collision by replacing bottle with pre-defined fragments and plausibly increase the water level as described in the technical outline.
2. Implement a bottle fluid shader to simulate water inside the bottle without requiring internal fluid simulation.
3. Add reflection & refraction to the water surface to enhance realism. Reflection can be done via multi-pass rendering, Screen-Space Reflections, or Cube mapping. Refraction can be approximated using Screen Space Refractions.
4. Add waves advection texture generated via FFT technique to the water surface. (See [2])
5. Add splashes and spray particles when a body interacts with water surface. (See [2])

# Bibliography


[1] J. de Vries, "Hello triangle," in *Learn OpenGL*. Kendall & Welling, 2020, ch. 5, pp. 26–41. [Online]. Available: https://learnopengl.com/book/book_pdf.pdf.

[2] N. Chentanez and M. Müller, "Real-time simulation of large bodies of water with small scale details," in *Proceedings of the 2010 Eurographics/ACM SIGGRAPH Symposium on Computer Animation, SCA 2010, Madrid, Spain, 2010*, Z. Popovic and M. A. Otaduy, Eds., Eurographics Association, 2010, pp. 197–206. DOI: 10.2312/SCA/SCA10/197-206. [Online]. Available: https://doi.org/10.2312/SCA/SCA10/197-206.

[3] "Lecture 13: Waves," University of Waterloo, CS 488/688, Spring 2025.

[4] J. de Vries, "Blending," in *Learn OpenGL*. Kendall & Welling, 2020, ch. 24, pp. 190–295. [Online]. Available: https://learnopengl.com/book/book_pdf.pdf.

[5] "Lecture 12: Particles," University of Waterloo, CS 488/688, Spring 2025.

[6] J. de Vries, *Learn OpenGL*. Kendall & Welling, 2020. [Online]. Available: https://learnopengl.com/book/book_pdf.pdf.