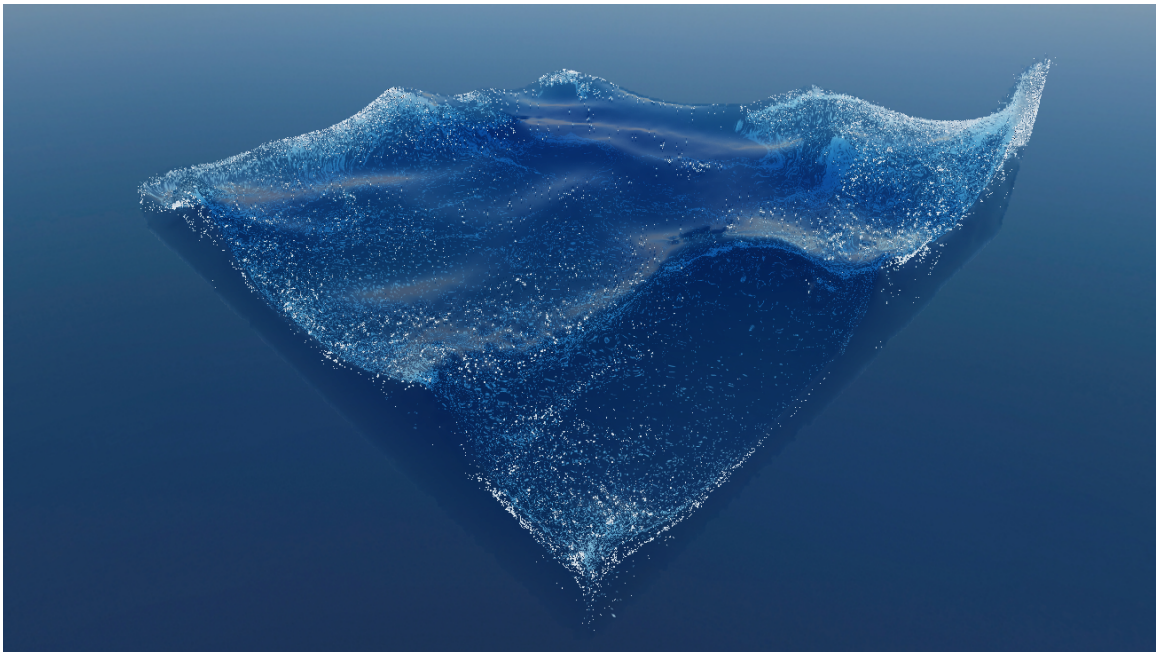
REAL-TIME RENDERING FOR FLUID SIMULATION

Final project submission for course: Real-time Graphics Programming (2026)



Luca Leopardi
Matr. No. 16080A

Università degli Studi di Milano, Dipartimento di Informatica “Giovanni degli Antoni”

Contents

Chapter 1

Introduction

This document details the implementation of a screen-space technique for real-time rendering of a particle-based fluid simulation, as part of a standalone Windows application.

Fluid simulation, detailed in a separate document, follows the Material Point Method (MPM) algorithm for material simulation. While MPM is a hybrid technique, utilizing both a particle- and a grid-based approach to model forces acting on and within materials, its final output is the integrated position of particles representing portions of mass. For the purposes of rendering, we can mostly ignore the workings of MPM and just assume that we'll be working on particles data, updated at each simulation step, representing each a uniform portion of fluid mass.

Rendering is done in OpenGL, using the GLFW, GLAD, GLM and ImGui libraries. OpenGL initialization, management and rendering of scene elements other than the fluid are done using standard techniques, and will be only briefly presented.

The goal of the project was to realize a system capable of rendering fluid at real-time performance of 60 frames per second, while maintaining a meaningful simulation size to visually showcase fluid properties and behavior. Relevant parameters for both simulation and rendering were exposed to the user interface, to allow for tuning of visuals and performance.

1.1 How to compile

- Install CUDA Toolkit (v12.9 or above).
- Install VS Code.
- Download Visual Studio Installer and install *Visual Studio Build Tools 2022 - C++ desktop applications* workload with components:
  - MSVC
  - Windows SDK
  - CMake tools
  - vcpkg package manager
- Install vcpkg via git and run setup script. No need for other setup steps.
- Install CMake (v4.0.3 or above).
- Install project's VS Code extensions (`.vscode/.extensions.json`).
- Edit `CMakePresets.json` file: `toolchainFile` must point to:
  `[VCPKG_INSTALL_FOLDER]/scripts/buildsystems/vcpkg.cmake`
  If CMake cannot find the Cuda Toolkit installation, manually set the cuda specifier in the `toolset` field to point to your Toolkit folder:
  `[...]/NVIDIA GPU Computing Toolkit/CUDA/[VERSION]`
  Set the `CMAKE_CUDA_COMPILER` field to point to:
  `[...]/NVIDIA GPU Computing Toolkit/CUDA/[VERSION]/bin/nvcc.exe`
- In VS Code, in CMake side panel:
  - Run Configure and select Visual Studio Build Tools 2022 Preset.
  - Run Build (Release or Debug preset).
  - Launch debugger or .exe in chosen preset's output folder.

1.2 Controls

The application presents a 3D scene with controllable camera, along with a toggleable user interface to modify simulation and rendering parameters.

Right mouse	Rotate camera
WASD	Move camera horizontally
Q/E	Move camera vertically
SHIFT	Fast camera movement
ENTER	Toggle user interface
ESC	Exit application

1.3 Testing methodology

All results in this report were captured on a machine with the following specifications, rendering to a 1920x1080 resolution window:

- **CPU:** AMD Ryzen 7 5700X3D (8 core | 3.00 GHz)
- **GPU:** NVIDIA GeForce RTX 3060 (12 GB VRAM) (CC 8.6)
- **RAM:** 16 GB 3200MHz
- **Storage:** SSD

Program benchmarks were captured using Nsight Systems, averaging frametimes and FPS over 5000 frames. Timings of specific passes was analyzed in RenderDoc captures. The main test scenario consisted of a simulation of 1 million fluid particles and 500,000 white-water particles in 100x100x100 MPM grid, rendering at a resolution of 1920x1080 pixels. A baseline performance value was measured by running the program with fluid rendering disabled, advancing only the simulation.

Chapter 2

Application structure

The application source code is organized into three main components, which will be presented in this chapter. Secondary structures and classes handling standard OpenGL functionality and UI might be mentioned but won't be explored.

2.1 Main

Application lifetime is managed in the `main.cpp` file. It performs, in order:

- GLFW window creation, setup and input binding;
- OpenGL context initialization;
- Simulation initialization;
- Scene setup;
- Application loop;
- Application shutdown;

During execution, in the application loop, `main` advances the simulation, passes its data to the renderer, draws the user interface, and handles any input.

The simulation advances at a fixed time-step hard-coded at 16.7 ms, corresponding to 60 steps per second. On each frame the delta time is measured and accumulated: if sufficient, it is consumed to advance the simulation by one step. The step rate can be set via the UI, effectively resulting in a speed-up or slow-down, but internally the simulation advances at the same 16.7 ms time-step. If total application frame-time exceeds the simulation's time-step it will not be able to provide data in real-time (indicated in-app by performance counters becoming red).

2.2 MPM Simulation

The fluid physics simulation is handled by the `MPMSimulation` class. It initializes the data structures for the particles and computation grid used in the MPM algorithm. The `step()` method is exposed for advancing the simulation, and CUDA kernels for parallel computation are defined.

CUDA-OpenGL interoperability allows to initialize data as OpenGL buffers, and then register and map it for use in CUDA kernels. After computation is finished, they can be unmapped and used for rendering. All data that is needed for visualization is handled this way:

- Fluid particles positions and velocities;
- Whitewater particles positions and types;
- Grid cells velocities and mass;

Data that is not needed for rendering is simply allocated as device memory using the CUDA API:

- Fluid particles velocity gradients and random generator states;
- Whitewater particles velocities;

Note that a Structure of Arrays data organization was chosen to improve performance in parallel computation. Memory-aligned data types have been used to optimize coalescence of GPU global memory accesses.

The buffers are sized to contain up to a hard-coded maximum number of particles or cells, with a count of the actual number of each being kept. VAOs are created for fluid particles, whitewater particles and grid cells, with the SoA buffers bound for rendering.

```
glGenBuffers(1, &_particles_positions_VBO);
glBindBuffer(GL_ARRAY_BUFFER, _particles_positions_VBO);
glBufferData(GL_ARRAY_BUFFER,
             MAX_PARTICLES_NUM * sizeof(glm::aligned_vec3),
             NULL, GL_DYNAMIC_DRAW);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
                      sizeof(glm::aligned_vec3),
                      (void*) 0);
glVertexAttribDivisor(1, 1);
glEnableVertexAttribArray(1);
cudaGraphicsGLRegisterBuffer(&_particles_positions,
                             _particles_positions_VBO,
                             scudaGraphicsRegisterFlagsNone);
```

Whitewater particles are also simulated, for added visual effect, though they are not considered in fluid stress calculations. They are spawned in areas of great fluid turbulence, and classified as either spray, foam or bubble depending on its local density. For advection, spray is affected by gravity, foam follows fluid direction, and bubbles rise according to buoyancy. Each whitewater particle has a starting lifetime (slightly randomized), which is reduced while in the foam state.

Unlike fluid particles, whitewater appears and dies as the simulation progresses. To keep live whitewater data compacted, their SoA buffers are actually split in two halves, and on each frame the surviving and newly spawned ones are copied from one half to the other. For this reason, the buffers are sized to contain double the maximum number of whitewater particles.

2.3 Renderer

A `Renderer` class handles drawing the scene to screen and the off-screen passes needed for fluid rendering. Separate `Model` and `Camera` classes contain data and logic for building the Model, View and Projection matrices used in the process. On application start it compiles shaders and initializes the 2D and cubemap textures needed.

The code and resources relevant to fluid rendering will be presented in the corresponding chapter. The rest of the `Renderer` class implements standard OpenGL forward rendering, as follows.

2.3.1 Models rendering

Opaque models, only used in the scene to showcase fluid reflections and refraction, are drawn using a simplified shader: their `Model` VBO defines vertex colors, and lighting is computed only as sum of the Lambert diffuse coefficient w.r.t. light direction and a generous ambient light constant factor.

2.3.2 Skybox

A skybox is rendered as background to the scene, by drawing a cube model centered at the Camera position, and using the interpolated vertex positions as 3D coordinates to sample a cubemap texture.

For the skybox to appear behind the scene, the easiest solution is to simply draw it before all other elements, with depth write disabled to not occlude them. To optimize, we can alternatively manually set skybox cube vertices $z = w$ in the vertex shader: after perspective division, all of its fragments will result at the maximum depth value of 1.0. This way the skybox can be drawn last, and only its non-occluded fragments will pass the depth test and be rendered.

2.3.3 Particles rendering

For simpler visualization and debugging, fluid particles can be rendered as just points or spheres, with options to color them based on their speed, velocity per axis, or basic diffuse shading. Point particles are drawn using OpenGL's `GL_POINTS` as primitive type, while sphere particles are drawn as "impostor" spheres on view-facing instanced quads (also known as billboards).

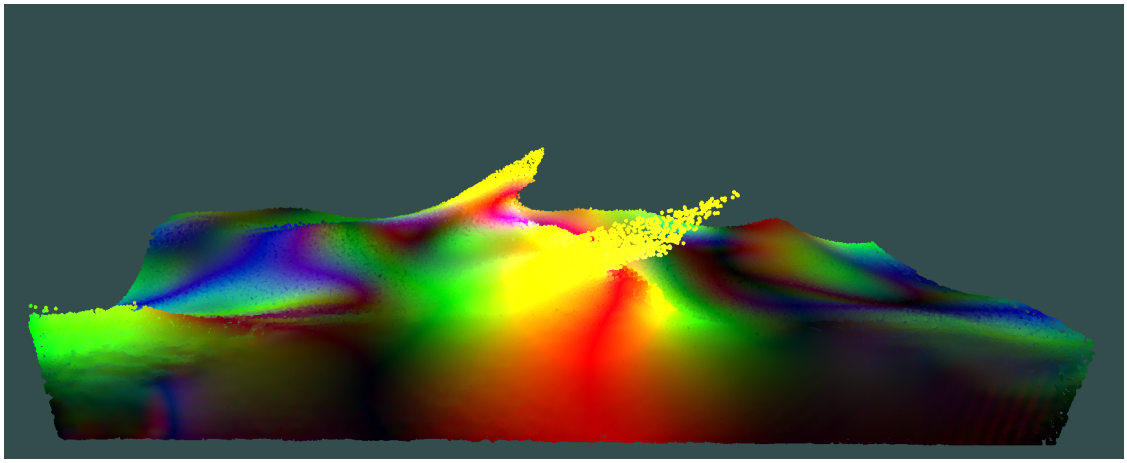


Figure 1: Particle visualization with color representing velocity along each axis.

In instanced rendering, the same VAO is drawn a number of times equal to the number of instances. Vertex attributes can be specified to stride per-instance instead of per-vertex, providing different data for each. We draw a single flat square mesh (a *quad*) for each of the simulation's particles, binding its SoA buffers as instance data. When rendering each, particle positions are transformed to world- and then view-space, at which point the quad vertex positions are added before moving into to clip-space. This results in drawing a view-space square at each particle position. A uniform can be passed to scale the offsets and change the effective particle size. The quad vertices offsets, once mapped from [-0.5, 0.5] to [0.0, 1.0], can function as UV coordinates in the fragment shader. A sphere impostor can be derived from UVs: mapped to [-1.0, 1.0] they correspond to the X and Y components of the sphere normal at the corresponding point. The squared radius and Z is then derived using the Pythagorean theorem. Fragments outside of the unit sphere are discarded.

```
vec2 normal_XY = v_UV * 2.0 - 1.0;
float r2 = dot(normal_XY, normal_XY);
if (r2 > 1.0) discard;
float normal_Z = sqrt(1.0 - r2);
vec3 normal = vec3(normal_XY, normal_Z);
```

Fragment depth is adjusted to accurately represent the surface position of the "inflated" sphere impostor.

```
vec4 view_space_pos = vec4(view_space_center + normal * particle_radius, 1.0);
vec4 clip_space_pos = projection_mat * view_space_pos;
gl_FragDepth = (clip_space_pos.z / clip_space_pos.w) * 0.5 + 0.5;
```

Chapter 3

Fluid rendering

The technique used for fluid rendering follows the algorithm outlined in the GDC 2010 presentation *Screen Space Fluid Rendering for Video Games* [1], with a number of adjustments and additions. The technique is summarized as follows:

1. **Thickness map pass:** Additively render fluid particles as circles, summing fragment values as an approximation of fluid thickness from the camera POV;
2. **Depth map pass:** Render fluid particles as sphere impostors, outputting linearized depth;
3. **Blur pass:** Blur the thickness and depth maps to approximate a unified surface from the individual fluid particles;
4. **Normals reconstruction:** Reconstruct view-space position from linear depth values and use partial differences between neighboring texels to derive surface normals;
5. **Compose:** Sample background scene applying normals perturbance to simulate refraction, apply tint depending on fluid thickness and light absorbance. Reflect view ray with normal and sample an environment map for reflections. Compute Fresnel reflectance coefficient and mix refraction and reflection accordingly.

`Renderer` exposes the `draw_fluid` method, which in turn calls functions responsible for each step. In addition to the algorithm as outlined, dynamic reflections have been added via environment mapping of the scene before fluid rendering. Whitewater (foam, spray and bubbles) has also been added by simulating and drawing additional particles as described in *Unified Spray, Foam and Air Bubbles for Particle-based Fluids* [2].

3.1 Environment mapping for dynamic reflections

Before the fluid is rendered, a cubemap of the scene as seen from the fluid's point of view is drawn, to be later used as an environment map for reflections.

It is rendered by pointing the camera along the six directions of the X, Y and Z axis and rendering the scene each time with a field of view of 90°. The resulting textures, mapped to a cubemap's faces, correspond to a 360° view of the scene from the camera's position.

When used for reflections, an environment map would only be accurate for surface points very close to its POV. So, parallax reflection correction is used to approximate the view from other surface points, making it usable for a larger area. With parallax correction, the reflected view ray is not used to sample the cubemap directly: instead, its intersection with a bounding volume (containing both the reflection point and the cubemap POV) is computed. The vector going from the cubemap POV to the intersection point is then used for sampling. The resulting correction is only and approximation, and its quality depends on a number of factors like the bounding volume shape, the distance to the reflected geometry, and others.

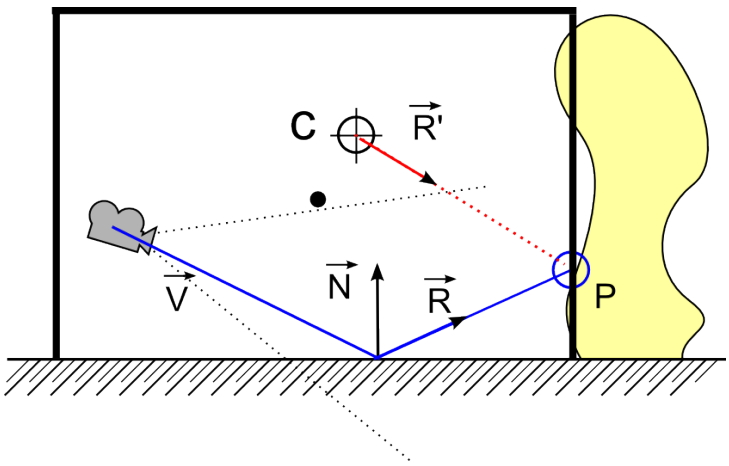


Figure 2: Parallax reflection correction for cubemaps (image by Sébastien Lagarde).

For our fluid reflections, an axis-aligned bounding box spanning the simulation bounds was used for parallax correction. The cubemap point of view is placed at the center of the simulation, with height adjusted to the estimated water surface level (approximated as the height of the fluid volume at rest, which is when reflections are more discernible).

To improve performance, each face is rendered at a square resolution of dimension equal to half the height of the screen. It is also only updated once every two frames. The limited reflectivity of the fluid, along with its turbulence, make these adjustments hardly noticeable.

3.2 Whitewater rendering

Whitewater is rendered as the last of the opaque geometry before drawing the fluid (but after the environment map capture), as we want it to be visible through it.

Particles are rendered as view-facing sphere impostors, using the same technique outlined in 2.3.3, with a couple of adjustments: whitewater initial sphere radius is half that of fluid particles, and is decreased proportionally to their remaining lifetime. Bubble whitewater particles have an even smaller size, and are rendered as circles instead of spheres, by discarding quad fragments both outside an outer and inside an inner radius.

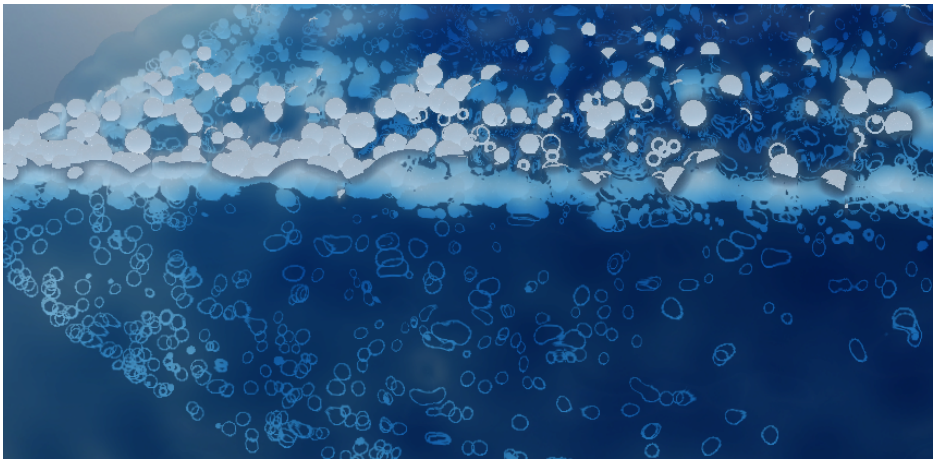


Figure 3: Final appearance of whitewater foam and bubbles.

Base whitewater color is pure white. This can make it look detached from the rest of the fluid, which reflects and refracts the scene background. For a more cohesive look, the skybox is used as an ambient light cubemap, sampled using the quad spheres normals. If viewed from up-close this results in an unrealistic, high definition diffuse reflection of the skybox image, but once mixed with the base white color, and from a normal viewing distance, it produces a subtle tint that makes whitewater feel more blended in the overall scene.

3.3 Framebuffer blitting and switch

So far all rendering has been done on the main framebuffer, in forward pipeline fashion. Once whitewater has been drawn, we move to an off-screen framebuffer for fluid rendering. The depth information of the scene in needed for accurate occlusion, and the image rendered so far will be used as the background to refract in the fluid: the depth and color buffers are copied (*blitted*) to the corresponding attachments on the off-screen

framebuffer. The depth buffer is also copied (downscaled) to a lower resolution depth attachment, used in the thickness pass.

During the fluid rendering process the off-screen framebuffer will be bound, with the appropriate texture attached as render target for each pass.

3.4 Thickness pass

As a first step in the actual fluid rendering, its thickness from the camera point of view is approximated by additively rendering each particle. At this step they can be drawn as simple billboard circles, summing a thickness value for each fragment drawn. Writing to the depth buffer is disabled so particles don't occlude each other, but depth testing is enabled to correctly represent thickness where opaque scene geometry (mostly whitewater) is in the fluid.

```
vec2 normal_XY = v_UV * 2.0 - 1.0;
float r2 = dot(normal_XY, normal_XY);
if (r2 > 1.0) thickness = vec4(0.0);
else thickness = vec4((2.0 - r2) * u_particle_thickness, 0.0, 0.0, 1.0);
```

The thickness map is rendered to a texture half of screen resolution to improve performance. A format of 16-bit float (stored in the red channel) was found to be sufficient to represent thickness data. Texture wrap was set to `GL_CLAMP_TO_EDGE` to minimize artifacts near the edges when sampling (valid for all textures in the following passes). The thickness map will be used in the composition pass to compute refraction and light absorbance.

Figure 4: Fluid thickness map.

3.5 Linear depth pass

For the next steps the fluid particles' linearized view-space depth is needed, which corresponds to the distance from camera. The particles are rendered as billboard spheres, this time with both depth testing and depth write enabled, as we need to draw only the surface closest to the camera. Fragments view-space position is computed as described in 2.3.3, and the Z value is saved as the linear depth map value. Actual fragment depth is still adjusted for impostors for correct depth testing.

```
linear_depth = vec4(-view_space_position.z, 0.0, 0.0, 1.0);
vec4 clip_space_position = u_projection_mat * view_space_position;
gl_FragDepth =  (clip_space_position.z / clip_space_position.w) * 0.5 + 0.5;
```

To obtain appreciable results, linearized depth requires more precision than thickness, so it is encoded as a full 32-bit float and rendered to a full-resolution texture. It will be sampled to reconstruct fluid surface view-space position and normals, and to apply depth-weighted blur.

3.6 Blur pass

In order to approximate a unified fluid surface, the depth and thickness maps are blurred. The blur shader works with a radius defined in world space (clamped between a minimum and a maximum) to try to maintain a similar level of detail both close to and far away from the camera. Actual blur kernel radius (in pixels) is determined per-fragment by computing pixels-per-meter at their view-space depth, by following the formula:

$$ppm = \frac{viewWidth * viewHorizontalScale}{depth}$$

The shader applies a gaussian blur kernel. The sigma parameter used to compute weights can be set via the UI. To maintain surface edge detail additional depth weighting is applied: this way fluid crests do no get blurred with the surrounding surface.

To improve performance, the blur is split into a horizontal and a vertical 1D pass. This introduces artifacts in depth-weighted blurring, but the effect is limited once rendering is complete. To reduce calculations per-fragment, the array of gaussian weights is also precomputed for the maximum blur radius, and samples are accessed symmetrically w.r.t. the center.

```
for (int i = 1; i <= radius; ++i) {
vec2 uv_dxdy = is_vertical ? vec2(0.0, i * texel_size.y) : vec2(i * texel_size.x, 0.0);

// Up/right sample
vec4 sample_value = textureLod(texture, v_UV + uv_dxdy, 0.0);
```

```
float weight = blur_weights[i];
if (u_blur_depth > 0.0) {
float depth_diff = depth - textureLod(depth_map, v_UV + uv_dxdy, 0.0).r;
float depth_weight = exp(-depth_diff * depth_diff / (2.0 * blur_depth * blur_depth));
weight *= depth_weight;
}
sum += sample_value.rgb * weight;
weights_sum += weight;

// Repeat for down/left sample...
}
```

Blur parameters and number of passes for the depth map can be set via the UI. The thickness map is also blurred (two passes only).

3.7 Normals reconstruction

Once we have an approximation of surface distance from camera, fragments view-space position can be derived from screen UV and linearized depth.

```
vec3 view_space_pos_from_UV(vec2 uv, float depth) {
    return vec3(
        (uv.x * 2.0 - 1.0) * depth / projection_mat[0][0],
        (uv.y * 2.0 - 1.0) * depth / projection_mat[1][1],
        -depth);
}
```

View-space surface normals can then be computed by using partial differences of neighboring texels' view-space position.

```
vec3 center_pos = view_space_pos_from_UV(v_UV, depth);

vec2 sample_uv = v_UV + vec2(texel_size.x, 0.0);
vec3 sample_pos = view_space_pos_from_UV(sample_uv, textureLod(depth_map, sample_uv, 0.0).r)
vec3 ddx = sample_pos - center_pos;

sample_uv = v_UV + vec2(0.0, texel_size.y);
sample_pos = view_space_pos_from_UV(sample_uv, textureLod(depth_map, sample_uv, 0.0).r);
vec3 ddy = sample_pos - center_pos;

vec3 normal_xyz = cross(ddx, ddy);
normal = vec4(normalize(normal_xyz), 1.0);
```

The resulting normals are saved to a color texture with the RGB color channels corresponding to the X, Y, Z components. To preserve numerical values in the [-1.0, 1.0] range, but make the resulting texture able to be rendered as color for debugging, internal texture format was set to `GL_RGBA8_SNORM`.

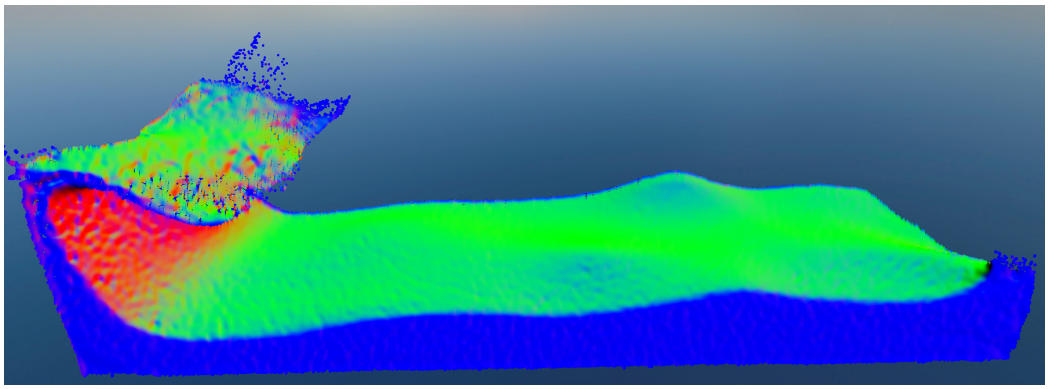


Figure 5: Surface normals resulting from the normals reconstruction pass.

3.8 Final composition

The depth, thickness and normals textures data is used to compute fluid refraction and reflections.

For refraction, the background scene texture is sampled using screen UV coordinates perturbed by the surface normal. The perturbance is scaled by fluid thickness and a customizable refraction strength parameter. While not a physically-based, this technique produces a convincing refraction effect that can be tuned with the given parameters.

```
refraction = texture(scene_tex, v_UV + normal.xy * thickness * refraction_strength);
```

To simulate the light absorbance of a ray passing through the fluid, Beer's attenuation law is applied with thickness representing the distance of traveled in the fluid.

$$I = I_0 e^{-kd}$$

Per-channel attenuation parameters are editable via the UI. The default values are a good approximation of water light absorbance.

```
float r = exp(-thickness * (u_light_absorbance.r));
float g = exp(-thickness * (u_light_absorbance.g));
float b = exp(-thickness * (u_light_absorbance.b));
vec3 light_decay = vec3(r, g, b);
refraction *= light_decay;
```

For reflections, the view ray is reflected by the surface normal in view-space. The inverse of the view matrix is then applied to transform the resulting direction into world-space, to be used as coordinates for cubemap sampling. Dynamic reflections are sampled from the scene environment map after applying parallax correction, as described in 3.1. If the sampled value is close to zero, reflections fall back to the skybox cubemap.

Refraction and reflection are blended according to Fresnel reflectivity, which describes how reflections are more pronounced when the surface is viewed at a more grazing angle. Shlick's approximation gives a way to compute the value from the index of refraction of the two materials and the half angle between incoming and outgoing light direction.

$$R_0 = \left(\frac{ior_1 - ior_2}{ior_1 + ior_2}\right)^2$$

$$R(\alpha) = R_0 + (1 - R_0)(1 - cos(\alpha))^5$$

In the shader, R_0 is pre-calculated with hard-coded indices of refraction of air and water, and the exponent in the formula is parameterized to tune the final effect.

```
float fresnel_0 = 0.02;
float cos_a = dot(-view_direction, normal);
float reflection_coeff = fresnel_0 + (1.0 - fresnel_0) * pow(1.0 - cos_a, fresnel_pow);
float refraction_coeff = 1.0 - reflection_coeff;
```

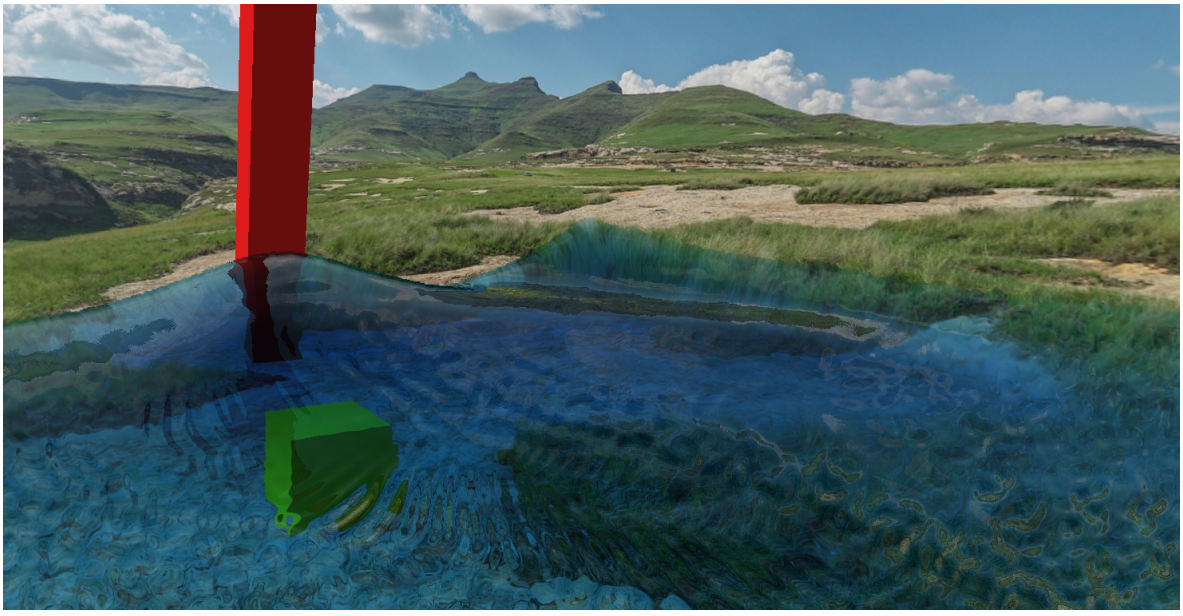
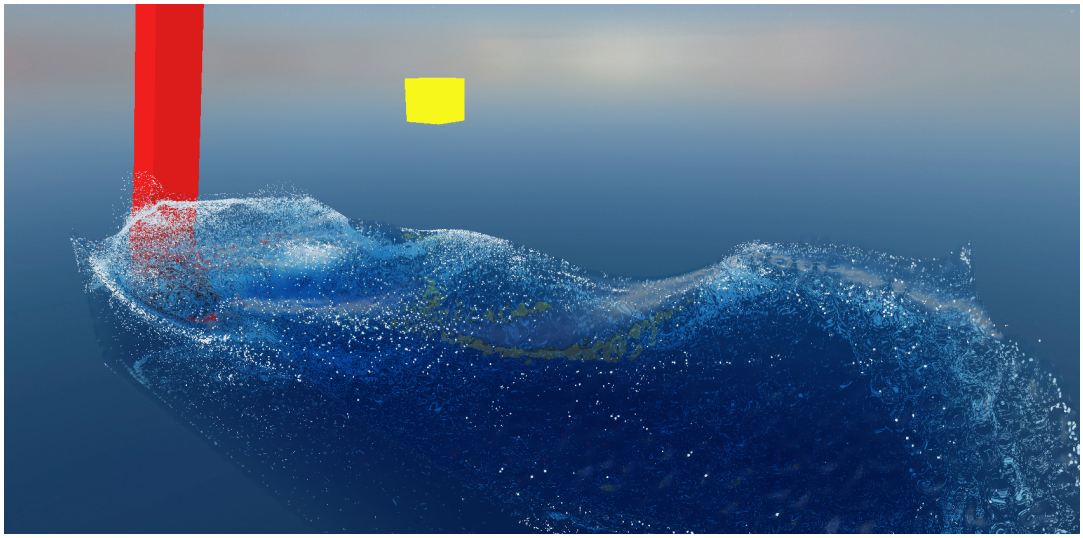


Figure 6: Reflections and refraction in the final fluid rendering.

3.9 Copy to screen

As a last step, the fluid is rendered to the default framebuffer by drawing a screen-spanning quad with the composition pass output texture bound. For debugging, intermediate passes can be selected in the UI: their output texture will be set for final copy instead.



Note that whitewater had already been drawn to the default framebuffer, and it is displayed correctly because its depth information was blitted to the off-screen framebuffer along with that of other elements of the scene.

Chapter 4

Results

4.1 Performance analysis

During development, RenderDoc capture timings were used to profile performance of the different passes. In the test scenario of 1 million fluid particles at 1920x1080 resolution, most of the time cost appeared to come from the thickness and depth map passes. This is to be expected, as they are the only ones to render actual particles.

The thickness pass was initially measured to take around 5.7 ms to render at full resolution. By halving the target resolution, timings were reduced to around 4.9 ms with minimal loss to visual quality. The pass still remained costly, as additive rendering with depth write disabled means that all particles are rendered, with no fragment being discarded early.

The fluid depth pass and whitewater rendering also weigh on performance, taking around 5.1 ms and 1.2 ms respectively. They render billboard sphere particles, where depth is added to flat quads: writing to `gl_FragDepth` disables early fragment discard, affecting performance. This cannot be ameliorated, as it is a key element of impostors rendering.

Blur passes (1D horizontal + vertical) are shown to take around 0.06 ms each on a full-screen texture (depth map), and 0.04 ms on a half-size one (thickness map). Thickness blur is hard-coded to two passes, while for depth the number of passes is exposed as a parameter that can be set in the UI. Four produce a good result for rippling water with a total time cost of 0.24 ms. Ten give a more tranquil surface water look with a cost of 0.6 ms.

Overall application performance was measured using Nvidia Nsight Systems, averaging frametime and FPS over 5000 frames. Multiple scenarios were benchmarked, first with just the fluid simulation running off-screen as a baseline, and then with rendering enabled. Across all tests the following performance-affecting fluid settings were kept: particle size = 0.6, depth blur passes = 4, blur kernel size = 0.35.

Fluid particles	Simulation only	Sim. + Rendering
32 000	0.5 ms (1548 FPS)	1.0 ms (665 FPS)
1 000 000	1.1 ms (860 FPS)	8.0 ms (124 FPS)
1 400 000	1.55 ms (646 FPS)	14.5 ms (68 FPS)
2 000 000	5.9 ms (168 FPS)	22.8 ms (44 FPS)

Table 1: Frametime benchmarks.

Results show that the application can maintain a real-time framerate of 60 FPS for up to 1.4 million fluid particles, in line with the goals set for the project.

4.2 Conclusions

The screen-space technique implemented produces good visual results and can handle a considerable simulation size while maintaining 60 FPS performance.

Nonetheless, numerous options for further visual improvement remain: 1D blur artifacts are still visible in certain conditions, though whitewater helps to hide them. Whitewater itself could also be improved with a number of post-process passes, mainly some form of blurring. Dynamic reflections are also limited, with parallax correction being insufficient on a large simulation area.

Bibliography

[1] Simon Green. Screen space fluid rendering for games. https://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf, 2010.

[2] Markus Ihmsen, Nadir Akinci, Gizem Akinci, and Matthias Teschner. Unified spray, foam and air bubbles for particle-based fluids. 28(6–8), 2012.