Fluid Simulation with Ray Marching

Gustav Nilsson Gisleskog[*] and Julius Thunström[†]

Lund University
Sweden

December 30, 2024

1 Introduction

In this project, a fluid simulation based on smoothed-particle hydrodynamics is implemented. One core idea behind this approach is that a particle is only affected by other particles within a fixed radius. Spatial hashing is used to efficiently query points within this radius, allowing a greater number of particles to be rendered each frame. Simulation steps are performed using compute shaders and instanced rendering makes efficient use of the computed positions. Finally, ray marching is leveraged to render a more realistic looking fluid.

The project was based on two excellent videos by Sebastian Lague. In [1], he develops a program for smoothed-particle hydrodynamics, and in [2] he uses ray marching and a number of other techniques to render the particles as a realistic liquid.

The source code for the project can be found at `https://github.com/aiviaghost/Fluid-simulation`.

2 Application

2.1 Physics

In order to behave like a fluid, the particles should of course be given properties based on the physics that governs fluids. As mentioned in the Introduction, Sebastian Lague has a video [1] on this topic. That video is the basis for this section.

2.1.1 Density and pressure

A good starting point is to define a density field over the box that the particles are in, and to let the density in a point depend on the distances to all particles. Formally, we define a *smoothing radius* r, such that particles outside the distance r from a point x have no effect on the density in x. Then we say that the contribution of a particle i, with position p_i, to the density in x is $S_D(\|p_i - x\|)$, where

$$S_D(d) = \begin{cases} \frac{6(r-d)^2}{\pi r^4}, & \text{if } d \leq r \\ 0, & \text{otherwise} \end{cases}. \tag{1}$$

The function S_D is called the *density smoothing kernel*. Finally, the density in x is the sum of the contributions of all particles:

$$D(x) = \sum_{k=1}^{n} S_D(\|p_k - x\|). \tag{2}$$

Next, we define a *target density* t, and try to make the density in each point equal to it, i.e. $D(x) = t$. The pressure in a point is defined as

$$P(x) = m_P(D(x) - t), \tag{3}$$

where m_P is a chosen constant. As such, the goal is to make $P(x) = 0$ in all points x by moving the particles. To do this, a *pressure force* should be applied to each particle p_i. Let d_{ik} be the distance between particle i and particle k, i.e. $d_{ik} = \|p_k - p_i\|$. We say that $f_P(k, i)$ is the force that particle k exerts on particle i, and define it as

$$f_P(k,i) = \frac{P(p_k) + P(p_i)}{2} \cdot S'_D(d_{ik}) \cdot \frac{1}{D(p_k)} \cdot \frac{p_k - p_i}{d_{ik}}. \tag{4}$$

Now, the force on a particle is the sum of the forces exerted on it by all other particles:

$$F_P(i) = \sum_{\substack{k=1 \\ k \neq i}}^{n} f_P(k,i). \tag{5}$$

From Newton's second law, $F = ma$, we can see that the acceleration should be $a = \frac{F}{m}$. However, in this program, the mass is not interesting since it is the same for all particles. Moreover, we view the particles as part of a contiguous fluid, which is what we are actually trying to model. Therefore, we replace the mass with the density when calculating the acceleration, which yields the following formula for the acceleration from the pressure force for particle i:

$$a_P(i) = \frac{F_P(i)}{D(p_i)}. \tag{6}$$

In each step of the simulation, the acceleration of each particle is multiplied by the time step length ($\frac{1}{60}$ seconds) and then added to the velocity. Subsequently, the velocity multiplied by the time step length is added to the position.

[*]e-mail: gisleskoggustav@gmail.com
[†]e-mail: thunstromjulius@gmail.com

2.1.2 Predicted positions

One thing to note is that when calculating the density and pressure, we actually use the *predicted positions* of the particles, rather than the actual positions p_i. The predicted position of particle i is defined as

$$p_i^{predict} = p_i + \frac{1}{60}v_i. \tag{7}$$

Doing this seems to make the fluid more "well-behaved".

2.1.3 Viscosity

In order to make the particles behave more like one singular fluid, rather than individual particles, viscosity is added. This adds a force that makes the velocities of neighboring particles more similar to each other. To do this, we first define the *viscosity smoothing kernel*:

$$S_V(d) = \begin{cases} \frac{4(r^2-d^2)^3}{\pi r^8}, & \text{if } d \leq r \\ 0, & \text{otherwise} \end{cases}. \tag{8}$$

Now, let the velocity of particle i be v_i. Then the viscosity gives it an additional acceleration $a_V(i)$, defined as

$$a_V(i) = \sum_{\substack{k=1 \\ k \neq i}}^{n} m_V S_V(d_{ik})(v_i - v_k), \tag{9}$$

where m_V is a multiplier for the viscosity strength. Similarly to the pressure force, this is added to the velocity of each particle in each simulation step.

2.2 Compute shaders

Our initial simulations used a single threaded CPU implementation to update the positions of all the particles. Since a lot of particles need to be rendered each frame to make the fluid seem plausible we quickly ran into the CPU being the bottleneck, even when switching to a multithreaded implementation.

To handle more particles we leverage the parallelism potential of the GPU to run these computations, specifically using compute shaders. Compute shaders are general-purpose shader applications that run separately from the regular graphics pipelines [3].

Before compute shaders can be used we first need to setup some memory locations to store the data on the GPU. For this we use so called shader storage buffers. The buffers only need to be allocated once at the start of the program and can then be reused for each simulation step. To make the code more readable we use structs, specified in the shaders, to aggregate information about the particles such as their positions, velocities, etc. To allocate space for these structs on the GPU in C++ there are some nuances when specifying the structs as the fields in the structs need to match the alignment of the structs in the GLSL code [4]. To solve this we had to add some dedicated padding-fields, as seen in Figure 1.

```
struct Particle {
    glm::vec3 position;
    float padding0;
    glm::vec3 predicted_position;
    float padding1;
    glm::vec3 velocity;
    float padding2;
};
```

Figure 1: Particle struct with padding-fields.

Each stage of the simulation process is handled by one or more compute shaders. To use a compute shader we first specify the shader to be used, set the necessary uniform variables and then call *glDispatchCompute* to specify the number of work groups and to actually invoke the shaders. Work groups are groups of compute shader invocations that, if desirable, can share data. As all steps of this simulation are designed to be easily parallelized we do not actually need to share data between these invocations. After invoking a specific shader we need to make sure all invocations of that shader are completed before running the next shader. This is accomplished using *glMemoryBarrier*.

2.3 Instanced drawing

Another bottleneck we ran into was issuing too many draw-calls to the GPU every frame. Our initial implementation looped over the particles and called *glDrawElements* for each particle separately. This results in a lot of overhead as drawing each sphere is quite fast but the overhead associated with the draw-call gets multiplied by the number of particles.

By instead using an instanced version of the draw-command, such as *glDrawElementsInstanced*, we can avoid this overhead [5]. This way we only issue a single draw-call every frame to draw all the spheres. The vertex shaders can then use their assigned *gl_InstanceID* to look up an offset to determine the position of a vertex. Specifically, the vertex shader will get this offset from the shader storage buffer containing positions. Note that these buffers always stay on the GPU so no data needs to be copied to or from the CPU when issuing the call to *glDrawElementsInstanced*.

2.4 Ray marching

At this point, we have a program that renders a number of spheres and makes their movements resemble those of a fluid. Now we want to make the spheres actually look like a real fluid. In Sebastian Lague's video [2], this is done with ray marching, and we follow his approach.

The idea is to, for each pixel, march along a ray and sample the density at fixed intervals to pick up color. As we follow the ray, the transmittance, which begins at 1.0 for all colors, decreases exponentially with the accumulated density and the scattering coefficients that are chosen for red, green and blue.

If the density in a point is positive, it is determined that it is in the fluid, and otherwise it is air. So, if we reach a point where we switch from air to fluid or vice versa, we hit the surface and the ray should be split into a reflection and a refraction. Technically, we should follow both of these new rays, but this would be too computationally expensive. Instead, we first approximate both rays by using a larger step size and no additional reflections or refractions. After leaving the bounding box of the fluid, they are sent into the environment, which in our case is a simple sky box. Then, the accumulated density of the two rays, together with the reflectance, is used to decide which of them is more "interesting". For the less interesting ray, the color we get from it is just calculated from its accumulated density, the transmittance that we had up until hitting the surface, and the color from the sky box. For the more interesting ray, we continue following it with the original step size, accumulate density and let it reflect and refract like the original ray. This process of approximating one ray and more closely following the other continues for a set number of times. Finally, if the ray is still in the bounding box after the maximum number of reflections or refractions, it is allowed to go directly into the sky box to get its color, just like the approximated rays.

3 Result

An example of the same particle configuration, using 2^{18} particles, rendered without and with ray marching can be seen in Figure 2 and Figure 3, respectively. We have tested the program on a few different GPUs, including a GTX 1070 Ti, an RTX 4070 Ti and an RTX 4080. Rendering the individual particles works quite well up to around 2^{18} particles while still maintaining a reasonable framerate when using an RTX 4070 Ti or better.

Enabling ray marching lowers the performance but often fewer particles are required when using ray marching to still get satisfactory visuals, compared to rendering the actual particles. This is partly due to the ray marching having a tendency to enlarge individual particles, which can be seen when comparing the leftmost side of Figure 2 and Figure 3.

There are some artifacts in the rendering using ray marching which can be seen for example along the vertical edge of the water on the right side of Figure 3. Too much light of various colors is reflected back towards the camera and this effect is amplified when increasing the number of reflections/refractions that we simulate. As of writing this report we have not found the underlying reason for this.

A minor issue with the physics simulation aspect is that when the splashing of the fluid is too intense or chaotic some clumps

[4] Christian Hafner. Compute Shaders. Available at: https://www.cg.tuwien.ac.at/courses/Realtime /repetitorium/VU.WS.2014/rtr_rep_ 2014_ComputeShader.pdf.

[5] Joey De Vries. Instancing. Available at: https://learnopengl.com/Advanced-OpenGL/Instancing.