Carbonated Drink Simulation

Akulov Dmitry, Amelichev Konstantin

March 26, 2024

Abstract

In the project we developed a carbonated drink simulation pipeline. It is suitable for simulating soda drinks or beer. Our code is available at github with several demo videos.

1 Introduction

This project is dedicated to beer/soda simulation. It combines 2 type of substance: liquid and foam/bubbles.
Liquid is simulated using SPH technique.
Foam is simulated using our own technique.
The code is based on cgp library.

2 Background on Simulating liquids

2.1 Stable fluids

Stable fluids is a grid-based method that can simulate dynamics in fluids that are static by themselves but react to external operations on gradient grid.

Stable fluids don't operate on fluid particles and don't usually work with internal forces, and thus they are not applicable in our case.

2.2 Procedural fluids

Procedural fluids are implicitly represented by function. Usually it is used to model surface. It is not applicable for our case, because we want to simulate behavior of bubbles inside of water, and it is not convenient to integrate explicit bubble instances to implicit reflection function.

2.3 Grid-based approaches

Grid based approaches wasn't suitable for our needs because they usually have static configuration. For us it was crucial that we have a higher precision in areas where there are a lot of bubbles.

2.4 SPh

SPh was a fit for our needs because it provides a particle-based system. It means that we can carbonate SPh-fluid by introducing new particles.

3 Environment

We've used a codebase from one of the assignments (Lab 12, sph liquids simulation). On top of it we've built our algorithm for simulating carbonated drink.

Researching chemical properties of carbonated drinks, we've discovered how the drink has gas with high pressure inside of a can/bottle. That's why we hear a sound when we open a bottle of soda.

3.1 Simulating liquid

For liquid we used sph method. This method is particle-based, where each particle stores a smooth kernel function that defines amount of water around a specific particle. For our implementation we use Gaussian kernel.

For kinematics on particles, we use forces, that correspond for weight, pressure and viscosity, as suggested in class. After forces are calculated, step of numerical integration is applied.

3.2 Spawning gas

For our SPh model the most natural way to spawn gas was through liquid particles. And indeed, if real-life drink is carbonated and releases gas when opened, the same behavior should exist for our simulation: liquid has paremeter of carbonacity and spawns gas until it is no longer carbonated.

We simulate gas as a separate particle. For performance considerations we don't account for forces between gas particles, but we calculate how it is affected by liquid.

Gas spawnage happens randomly, every frame there is a chance liquid will spawn a gas particle and reduce it's carbonacity.

To avoid modelling complex real-life situations where liquid might be moving around glass or be poured somewhere, we've decided not to spawn gas particles in these cases. After all, when liquid is moving, you mostly don't see particles in real life. So, we spawn gas particles randomly and only when there is no strong forces influencing liquid particle.

3.3 Gas kinematics

We know from physics, that gas molecules are floating from the water either because of different densities. There is a pressure applied to the bottom of a bubble until it floats. We simulate this with constant force that pushes bubble out of the water.

At the same time, when bubble is at the surface level, it doesn't necessarily disappears and merges with air. There is also a surface tension force that traps gas near surface. Because of that there might be either a foam or sparkling effect. For us it means that surface location is special.

To gather bubbles around surface level, we use a hack with another force. With our currently described kinematics carbon dioxide will behave as a soap bubble, floating out of water. It is a valid behavior for soap water, because it has higher surface tension, but isn't what we expect from a regular liquid. To address this, we add a gravity force to bubbles, if they aren't in contact with water. With these two forces balanced, bubbles tend to gather slightly above surface level.

3.4 Popping Bubbles

We have 2 options for popping bubbles. It depends on the checkbox "More foam".

Setting False, the simulation resembles soda.
In this case bubbles pop randomly. Each one has equal probability of being popped. They disappear faster.

Setting True, the simulation resembles beer.
In this case a bubble has a higher probability of being popped if it's higher and if it has low number of neighbours. This logic supposes that the top layer of bubbles pops.

4 Foam Physics

When bubbles are gathered together, they form a foam. The structure of a foam is different from a regular set of bubbles, because there is not enough room for all bubble's spheres. Basically, it is a two spheres of gas, that have a water membrane between them under pressure.

So, our grouping strategy for bubbles is the following: usually, every bubble is a sphere, however if bubbles are too close together, they can't both be spheres, because then they would overlap. So, they have a membrane between them.

5 Drawing algorithms

For displaying simulation results, we had several challenges. To begin with, we had to come up with drawing bubbles and foam. After that, we encountered performance issues for default sph rendering and optimized it.

5.1 Optimizing drawing Fluid

After simulation step for sph fluid is done, we have a set of particles and Gaussians linked to them. Original code base was applying corresponding Gaussians globally. However, several units away from particle, effect becomes negligible. To optimize the drawing process, we reversed it. Now instead of going through all pairs of cells and particles, we only iterate over close neighbors.

To do that, we had to inverse iteration order. Now we start with selecting particle, go over small adjacent area, increase color response over the area. At the end, we go through all the pixels to accumulate results.

5.2 Drawing Bubbles

For bubbles we wanted effect similar to sph, so we use the same gaussian approach. For center of gaussian, we manually fill bubble with white color, so it is more distinct.

5.3 Drawing Foam

Sparkles form a special unique structure.
We found that it resembles Voronoi diagram.

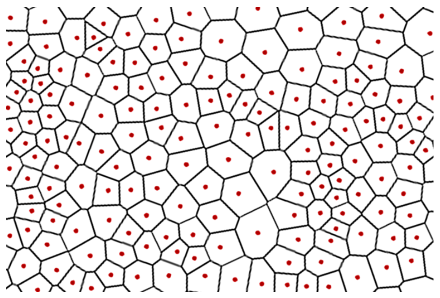

Figure 1: Chordes for Voronoi

Our goal was to simulate something similar to Voronoi diagram with respect to the limits of bubbles' volume.

5.4 Voronoi Diagram Algorithm

Constructing Voronoi Diagram is a well-known yet complex algorithm.
There are different ways to construct it, with complexities from $\mathcal{O}(n^4)$ to $\mathcal{O}(n)$ on average.
We start with the basic one($\mathcal{O}(n^4)$).

Let's describe some properties.

- Each segment of the diagram is a part of the perpendicular bisector of a pair of points.
- Each vertex of the diagram is the center of the incircle of some three points.

The basic algorithm works the following way.
Let's choose some point i. We draw the perpendicular bisector between p_i and p_j for all j. Then we intersect all the perpendicular bisectors, getting $O(n^2)$ points. Then we can check that each one of that points lies at the correct side from each of n half-planes.
It takes $\mathcal{O}(n^3)$ to build one cell. So, to build cells for all points it would take $\mathcal{O}(n^4)$.
$\mathcal{O}(n^4)$ is too much when n exceeds 100 (and that's the case for us).

5.5 Adapted Voronoi Diagram Algorithm

Due to specifics of the environment, our algorithm works in $\mathcal{O}(n^2 + nk^2)$, where n is the number of bubbles and k is the empirical upperlimit on the number of bubbles one bubble can touch.
In our case $n \leq 400, k \leq 20$.
So, to render one image it takes around $300'000$ operation. In c++ this amount of float operations fits into $0.0003s$, with other constant complication it shouldn't take more that $0.01s$, so the fps shouldn't be affected.

We start the algorithm with intersecting each pair of bubbles. We get chords that connect 2 points by which the bubbles intersect. Those chords are parts of the perpendicular bisector. And parts of that chords will be edges of our Voronoi Diagram. It takes $\mathcal{O}(n^2)$.
Let's again choose some point i. Since each bubble touches at most $k \leq 20$ other bubbles, we are left with at most k chords. Now we have to cut each one of them.
To cut a chord, we intersect it with each other chord and update the left or right end of a chord.

So, it takes $\mathcal{O}(k^2)$ operations for one bubble and $\mathcal{O}(nk^2)$ in total.
The result may look like Fig. 2. Cut parts are shown with red, remaining with green

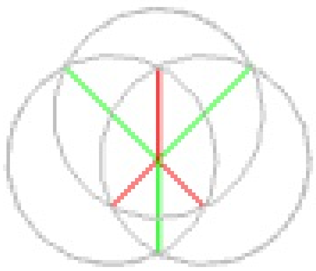

Figure 2: Chordes for Voronoi

When we know Voronoi segments, we also have to draw arcs. Since cgp library can draw only complete circles we have to do it by ourselves. We do it by approximating an arc with several chords. For an indistinguishable picture we can draw a chord approximately every 20 degrees
Eventually that 3 bubbles look like Fig. 3.

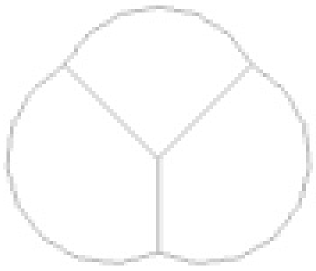

Figure 3: 3 Bubbles

Code can be found in the function `scene_structure::display_voronoi` in the file `project/src/scene.cpp`.

Fig. 4 is an example of what our foam looks like.

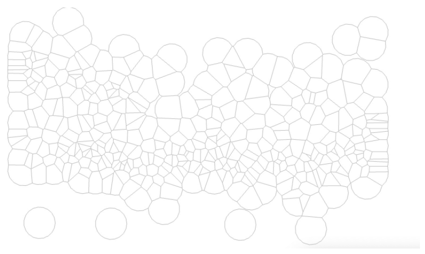

Figure 4: Many Bubbles

6 Results

During the project, we developed an algorithm for simulating carbonated drinks. We've implemented simulation of bubbles and a method for displaying a foam structure based on Voronoi diagram.

Resulting simulation program has several configurable parameters, that either allow beer-style foam or soda-style top layer of bubbles.

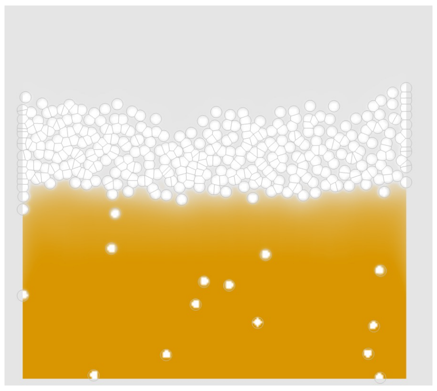

Figure 5: Beer simulation

Current limitation is that our algorithm doesn't merge bubbles together – though, we believe, this is possible.

Another limitation is spawning process: if bubble is spawn too high, it will take too long to fall back and it looks unnatural. We've modified initialization so it influences not so much particles, but it is still a thing to improve.

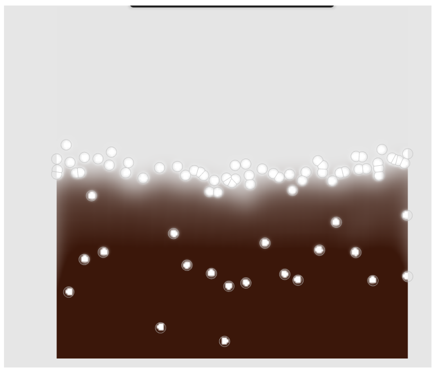

Figure 6: Cola simulation

We have several demo videos of this project at our github page.