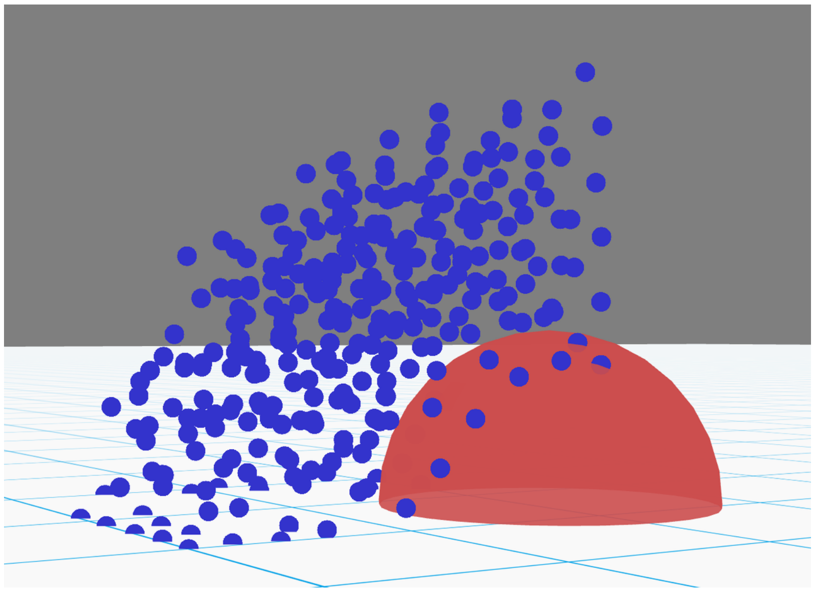
Assignment 2: Position-Based Fluids

Due March 2nd at 11:59pm



Introduction

Fluid simulation is a common challenge in computer graphics, and one that illustrates the need for efficient physical models. While physically accurate simulations of fluid dynamics can be conducted by solving the Navier-Stokes equations exactly, this is computationally expensive in general, and as such is often not suitable for graphics applications, especially those that are meant to run in real time.

Therefore, an active area of research has been to search for more efficient discretizations of fluids. Particle-based simulations are attractive in this regard due to their relative simplicity. In this assignment, you will be implementing one such particle-based simulation, based on the paper "Position-Based Fluids" by Macklin and Müller (accessible at `http://mmacklin.com/pbf_sig_preprint.pdf`).

Getting Started

The structure of the codebase is the same as in the previous assignment, the only difference being that the `Playground` directory has been replaced by the `Assignment2` directory containing the new starter code. Please follow the same steps as in the previous assignment to compile the code. On Windows, you may be able to save some time by simply taking the project in `Assignment2` and adding it to your existing Visual Studio solution.

As before, the places where you will need to fill in code are marked with a `TODO`.

Algorithm Overview

We have provided you a basic particle simulator that will integrate velocities over time. Note that we are no longer computing accelerations from forces; rather, forces are applied by adding them directly to velocities. This is because we are assuming that all particles have unit mass, so the acceleration at each timestep will simply be equal to the force (scaled by the step size). Your primary task is to implement the algorithm described in "Position-Based Fluids". In this section, we give a brief overview of some of the key ideas behind the algorithm.

The paper falls into a class of techniques in fluid simulation known as "smoothed particle hydrodynamics" (SPH). As the name suggests, these techniques involve discretizing water into particles, and then using a smoothing kernel to "smear" the mass of the particle out into the surrounding area. In this way, the particles define a scalar field at each point in space, representing the density of the fluid, and the density at any point can be computed by adding up the contributions of all particles within distance h, where h is the radius of the kernel.

Fluid-like behavior is then achieved by defining some rest density ρ_0, and imposing a constraint on all particles that the density ρ_i at any given particle should be equal to the rest density ρ_0. The term "position-based" refers to the fact that we enforce the constraints by directly manipulating the positions of particles, rather than deriving a force from the energy function of the constraint via $-\frac{\partial W}{\partial x}$, as we have been doing in class. At each timestep, we compute a positional correction Δp_i that brings the particle closer to satisfying the constraint (i.e. such that $C(x_i + \Delta p_i) = 0$). We then directly update each particle's position by $x_i \leftarrow x_i + \Delta p_i$.

The algorithm described in the paper can be roughly broken up into a few key components which you will have to implement:

- **Collision handling.** At a minimum, you will have to keep the fluid inside the box by implementing collisions with planes. This can be done by correctly implementing the collision function in `CollisionPlane.cpp`.
- **Neighbor finding.** The constraints on each particle are defined with respect to their sets of neighbors. Each timestep, for each particle, you will have to find all the particles that are within distance h (the kernel radius) of it. We have already provided a basic spatial hashmap to do this, so you do not need to do any work for this part.
- **Density constraints.** As discussed, the constraint on the density at each particle is what causes the fluid to resist compression.
- **Surface tension.** The paper introduces a correction term s_{corr} to avoid excessive clustering of particles. As a side effect, this also approximates surface tension.
- **Vorticity confinement.** This helps to counteract the effects of numerical damping in the system, by computing the vorticity (roughly speaking, the rotational motion

about a point) at each particle, and applying an additional force based on how much vorticity is present.

- **XSPH viscosity.** This essentially attempts to smooth out differences in velocity between neighboring particles, causing the particles to flow more coherently instead of undergoing random independent motions.

You should refer to the paper for the exact details on how to implement these components.

The skeleton code we have provided is less complete than in the previous assignment, in order to allow you to structure your code in whatever way you are most comfortable with. It may be a good idea to split up these parts into one or more separate functions for each. You are free to edit the header files in order to do so.

The primary data structure you will be working with is the struct in `Particle.h`; this contains all the fields that are necessary to describe a particle's state. The particle system keeps a list of these structs, and you will need to edit the fields of these particles in order to reflect state changes.

Density Kernels

Throughout the paper, you will see the notation $W(\mathbf{p_i} - \mathbf{p_j}, h)$, which refers to the density kernel with radius h evaluated on the vector $\mathbf{p_i} - \mathbf{p_j}$. There are two kernels in use in the paper. One is known as the "poly6 kernel", which is defined as:

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

The poly6 kernel is used for evaluating the density at a particle. Note that there is a slight abuse of notation; the (non-bolded) r in W_{poly6} really refers to the magnitude of the input vector $\mathbf{r}$.

The other kernel is known as the "spiky kernel", which is defined as:

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

The spiky kernel is not used for density estimates, but is only used for computing the gradient of the density. In this case, we use the gradient of the kernel:

$$\nabla_{\mathbf{p_i}} W_{spiky}(\mathbf{r}, h) = \frac{-45}{\pi h^6} \begin{cases} (h - r)^2\, \hat{\mathbf{r}} & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

Here we assume that $\mathbf{r} = \mathbf{p_i} - \mathbf{p_j}$, and $\hat{\mathbf{r}}$ is the normalized direction of $\mathbf{r}$. If instead we want the gradient with respect to $\mathbf{p_j}$, we have to flip the sign due to the chain rule.

In general, whenever the kernel $W(\mathbf{p_i} - \mathbf{p_j}, h)$ appears, you should evaluate the poly6 kernel. Whenever the kernel gradient $\nabla W(\mathbf{p_i} - \mathbf{p_j}, h)$ appears, you should evaluate the gradient of the spiky kernel. You should never need to evaluate the spiky kernel itself.

A note on constants

All constants necessary to the algorithm are contained in `Constants.h`. For instance, the kernel radius h in the paper is defined by `KERNEL_H`. It is recommended for consistency that you use these constants in your implementation of the algorithm. The values have already been set to produce decent behavior on the examples included with the release. However, you may change them to whatever you want in order to achieve different effects.

Extension

Once you have completed the simulator, you should extend it to produce a creative artifact of your choosing. Some ideas include:

- **Better spatial acceleration.** The spatial hashmap we provided performs well when particles are spread out in space. However, once particles have settled to the bottom of the container, they will all be concentrated in the same few grid cells, lowering performance. You may want to implement a more advanced data structure that gives better performance even when particles are concentrated in the same area.
- **Rigid bodies.** At the moment, the water only interacts with itself and with the boundaries of the box. You could liven things up by adding different objects inside. You could start with simple geometric objects like spheres and cubes, and move to more complex ones.
- **Different scenes.** Rather than sticking with the box, you could try pouring water over different environments. You could start by taking the existing `CollisionPlane` class and simply adding different planes than the ones that define the box. However, the range of environments you can achieve with this is also quite limited. You might consider importing height fields or arbitrary meshes as obstacles.

These are just suggestions – you're also free to pursue an idea of your own. Feel free to talk to the professor or TA about your ideas.

Submission instructions

You will need to submit three parts: a write up, your code, and a submission video. When submitting your code, please delete the `Assignment2/Release` and `Assignment2/Debug` directories if they are present. Then, zip **only** the `Assignment2` directory.

If you have created additional examples for your extension, you may also zip those and send them to us.

To export a submission video from your project, you can use the "Record Screenshots" checkbox at the top-left of the project interface. Once checked, the program will write each frame of the simulation to an image in the `screenShots` folder in the project directory. Note that it will be writing uncompressed `.bmp` images for each frame, so the size of this directory can grow quite large – make sure you have sufficient disk space before doing this. Additionally, this will cause the simulation to become quite slow as it writes the images, though this will not affect the speed of the video.

Once you have exported the series of frames, you should be able to combine them into a video using most video editing software. VirtualDub is a free tool that also has this functionality. Alternatively, if it is more convenient, you can use a screen recording tool to capture the window directly.

In order to submit, please e-mail the following items to the instructor (scoros@cmu.edu) and TA (christoy@cs.cmu.edu):

- A brief writeup (2 pages max) that describes your effort in implementing this assignment, including the extension you have implemented, what worked and what did not work, and any interesting insights you may have gained.
- A `.zip` of the contents of your `Assignment2` directory, after having deleted the `Debug` and `Release` directories inside (please be careful not to delete your code!).
- A `.zip` of any additional data files you have created for your extension, if applicable.
- Either your submission video, or if it is too large, a link to somewhere we can download your submission video (e.g. Youtube, Dropbox, etc.).

Please also include the string "CS15467" in your subject line so we know to look for it.

Notes on Academic Integrity

You are allowed to collaborate on the assignment in terms of formulating ideas, developing physical models and mathematical equations. However, you must implement the code and do the write up completely on your own, and understand what you are writing. Please also list the names of everyone that you discussed the assignment with.