Ice Ice Baby: A Temperature Dependent Water Simulation

Alex Schedel
alexschedel@berkeley.edu

Lucy Lu
zhixinlu@berkeley.edu

Olivia Huang
ohuang@berkeley.edu

Eddy Byun
eddybyun@berkeley.edu

1 ABSTRACT

Simulations are essential to modern computer graphics. They allow for physically accurate simulations in fully virtual space, adding realism to films, video games, and other digital images. Fluid simulations in particular are essential for adding realism and often employ abstractions that enable the simulation of millions of water particles in a fast and easy way. Something that is less commonly discussed, however, is simulating fluids as they transition between states. The computational models that are optimized for fluid simulation may no longer be applicable when that fluid turns into a solid. In this project, we aim to implement a novel particle-based simulation in order to model the state transition of water to ice, in effect "freezing" our simulation in a realistic-looking way. To solve this issue, we present Ice Ice Baby, a temperature-dependent water sim.

2 INTRODUCTION AND BACKGROUND

When simulating and rendering fluids, it is often useful to model the physical nature of the fluid as a more abstract representation, such as a fluid grid. By removing the individual particles of the fluid and instead modeling the forces that change large swaths of fluid, grids can act as an acceleration structure, enabling faster rendering for large fluid sims. However, design flexibility is also lost by removing individual particles, in particular if it is important for individual particles of water to react with each other, a more expressive design is required. This problem becomes apparent when simulating state transitions of various fluids. Simulating the transition from water into ice is not easily compatible with a fluid grid model as the colder the water becomes, the more it behaves like a series of particles and the less accurately a grid of forces would be able to capture its behavior.

2.1 Particle Based Water Simulations

Inspired by the work of Matthias Müller, David Charypar, and Markus Gross[3] and unlike many water simulations, Ice Ice Baby employs a particle-based simulation. In terms of simply simulating water, a particle based implementation provides no real advantage over a conventional grid based implementation. However, as the simulation transitions from water into ice we leverage the organizational structure of a mass and spring-based system, wherein the masses are the particles, to allow the water to appear to clump together and behave more like a solid. While this would be difficult to achieve with a fluid grid-based model, our particle-based simulation offers a natural jumping-off point for this extension.

2.2 Mass and Spring Systems

Mass and spring systems are often used to model movable meshes, such as cloth and hair simulations. Masses are used as anchor points and the springs are used to influence the movement of those anchor points relative to each other. However, because water does not consist of discrete particles (at least at the scale of human perception) they are rarely applied to fluid simulations, as fluid sims often forego representing individual particles in favor of representing forces and currents. Although our simulation does not utilize springs for the purpose of exerting force, we uses it as an organizational tool to characterize the way that particles are related to and interact with each other.

3 ICE ICE BABY

3.1 Our Approach

This simulation is built off the architecture provided by CS284a project 4, which implements a mass and spring-based cloth system. Additional equations and algorithmic inspiration for the particle-based simulation are inspired by the work of Miles Macklin and Matthias Muller [3] with additional equations inspired by Matthias Müller, David Charypar, and Markus Gross [2].

Fundamentally, we have implemented and extended the Macklin and Muller algorithm[3] to account for our spring system as well as the correction vectors entailed by it. Our extension of the algorithm is as follows:

Algorithm 1 Simulation Loop

for all particles i **do**
  apply forces $v_i \leftarrow v_i + \Delta t f_{ext}(x_i)$
  predict position $x_i^* \leftarrow x_i + \Delta t v_i$
end for
build spatial map of particles i
for all particles i **do**
  find neighboring particles $N_i(x_i^*)$
end for
if water is freezing **then**
  for all neighbors n_i of particle i **do**
    create spring $s_{i \to n_i}$
  end for
end if
while $iter < solverIterations$ **do**
  for all particles i **do**
    calculate λ_i
  end for
  for all particles i **do**
    calculate Δp
    detect collisions on i and respond
  end for
  for all springs s connecting particles i and j **do**
    calculate correction c
    $x_i^* \leftarrow x_i^* + direction * c * 0.5$
    $x_j^* \leftarrow x_j^* - direction * c * 0.5$
  end for
  for all particles i **do**
    update position $x_i^* \leftarrow x_i^* + \Delta p_i$
  end for
end while
for all particles i **do**
  update velocity $v_i \leftarrow \frac{1}{\Delta t}(x_i^* - x_i)$
  update position $x_i \leftarrow x_i^*$
end for

Our simulation consists of two main elements, which shift in importance as we change the temperature of our simulation.

The first of these elements is our particle-based water simulation. Each particle represents a certain amount of water. Water is by nature incomprehensible, meaning that it is essential we constrain these particles to maintain minimum distances from each other, even if large amounts of pressure are pressed upon them. To this end, we implement the Navier-Stokes Equations[4] as well as several gradient and kernel functions to simulate impressibility. These additions alone are enough to begin making the water simulation appear at least somewhat realistic, even without the particles yet being rendered as a continuous amount of realistic water.

The second element is our organizational mass and spring system. The simulation draws springs between individual particles. These are not springs in a traditional sense as they do not exert forces directly on each other, rather they are used to calculate correction vectors that ensure that neighboring particles never get too close or far from each other. Our simulation begins with all particles fully disconnected, meaning there are no springs in the model. As we simulate a temperature decrease and an eventual transition into ice, the allowable difference in distance between neighboring particles and the spring rest length decreases, ultimately stopping movement altogether once the simulation reaches specified "freezing" temperature.

3.1.1 Code Reorganization and Extension. The project itself uses the CS284a cloth simulator as a jumping-off point.

Our first changes to this base code came in how we represented points. The base code represents points simply as having a position and forces which act upon them. The algorithm above however requires a much more expressive notion of a point. To this end we added in variables to represent the velocity, lambda, and neighbors of each particle.

Beyond the change in particle representation, our next change came in the simulation function. The starter code implements basic cloth simulation, which we refactored and extended to be in line with the algorithm above. This was where the bulk of the programming and testing occurred.

We also added new primitives to the simulator in order to test as well as demonstrate our progress. By default, the starter code contains the ability to render a cloth, a plane, and a sphere. We added the ability add a "container" which would allow us to demonstrate the water spreading out and collecting in a single surface. Additionally, we implemented transparency and the ability to scale planes and spheres to make sure that we would have a large amount of freedom in creating simulation environments.

Finally we extended the spatial map from the original project to look not only at the grid a given particle is in, but the surrounding grids as well. This allows us to take nearby particles into affect while ignoring distant particles which are unlikely to affect the one we are looking at.

3.1.2 Kernels and Gradients. In order to realistically simulate fluid dynamics and the movement of water, we used a position correction calculated by Macklin and Muller [3] that allowed for the particles to have a clumping effect and maintain a consistent density. We use the poly6 kernel [1] in order to calculate the density of the particle and smooth out the position in relation to its neighbors, which is defined by the following equation:

$$W(\mathbf{r}, h) = \frac{315}{\pi * h^9} \left\{ \begin{array}{ll} (h^2 - r^2)^3 & 0 \le r \le h \\ 0 & otherwise \end{array} \right\} \quad (1)$$

We first calculate the constraint functions and scaling factor for each particle, defined as the following:

$$C_i(p_1 ... p_n) = \frac{\rho_i}{\rho_0} - 1 \tag{2}$$

where

$$\rho_i = \sum_j m_j W(p_i - p_j, h) \tag{3}$$

We iterate over each neighbor j, calculated by storing the neighboring particles in a position-based hash table, called a spatial map. The mass m_j is set to 1 for each particle. ρ_0 is the rest density is set to 45000, and $h = 0.03$. We then calculate the gradient of the constraint function with respect to the particle p_k as

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \begin{Bmatrix} \sum_j \nabla_{p_k} W(p_i - p_j, h) & if k = i (itself) \\ -\nabla_{p_k} W(p_i - p_j, h) & if k = j (neighbor) \end{Bmatrix} \tag{4}$$

With these two values, we can finally calculate the scaling factor λ_i with offset $\epsilon = 0.1$

$$\lambda_i = -\frac{C_i(p_1 ... p_n)}{\sum_k |\nabla_{p_k} C_i|^2 + \epsilon} \tag{5}$$

We then finally calculated the position correction Δp_i as

$$\Delta p_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(p_i - p_j, h) \tag{6}$$

where the gradient of our kernel was the gradient of the spiky kernel, defined by

$$W(\mathbf{r}, h) = \frac{15}{\pi * h^6} \begin{Bmatrix} (h - r)^3 & 0 \le r \le h \\ 0 & otherwise \end{Bmatrix} \tag{7}$$

This Δ_p was applied to the position vector of each particle after calculating the new position in the timestep of our simulation.

3.1.3 Spring System and Freezing. Although we use the existing infrastructure of the base code for our springs, the way we use the springs to implement particle freezing is very different to what was done in project 4. To begin, we connect every particle with its neighbors at every step of the simulation. Next we calculate correction vectors for the position of each particle. If the simulation is fully unfrozen, this correction vector will be 0. If it is not, a correction vector will be generated and used to modify the position of the particle. As the temperature of the water gets closer and closer to freezing, this correction vector will become more and more strict, meaning that it will appear to more and more limit the movement of the particle, reducing the amount of movement in the system and ultimately simulating freezing.

This correction vector does need to be slightly reigned in however. The vector is applied according to newtons third law, meaning that each particle at the end of the spring that is determining the correction will be moved back by half the vector. This means that it is possible for a single particle to be pushed forward by neighboring particles even when it should be moving backwards. To correct this we manipulate the minimum length of our spring to ensure that particles will only get farther away from each other rather than closer. This allows the springs to appear to resist contraction, allowing our water to more realistically appear to spread out.

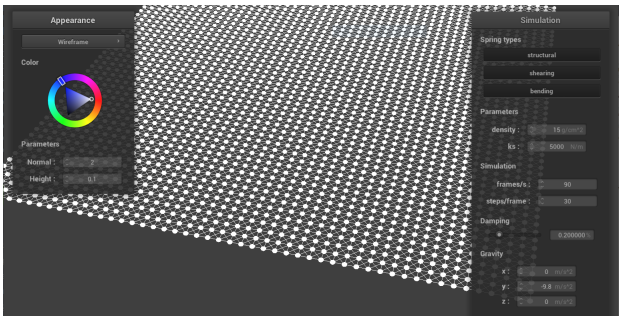
3.2 Texturing and Rendering

While starter code does provide a GUI and method to simulate, it has limited functionality that we had to extend. The base code not actually render any of the points on screen, opting instead to only show the springs that connect them. This meant that we had to add additional logic to the cloth simulator itself that would allow us to render and rerender particles. We ended up rendering the particles as transparent sphere and decreasing the transparency as the temperture of the simulation dropped. This allowed us another way to see that the particles were freezing aside from them simply lacking movement.

As for rending our final gifs and images, we found that as the simulation became bigger it also quite quickly became difficult to render in real time. To fix this we rendered our final gifs by recording the simulation in real time, taking each image from that simulation as it rendered, and stringing them together after the fact into a single continuous gif. Smaller simulations and tests however can be rendered in real time.

3.3 Testing

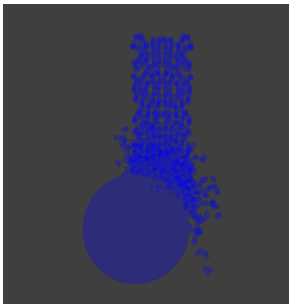
3.3.1 Early Tests. Initial testing largely revolved around getting particles to render on the screen at all. As noted above, the base code does render points, so we had to figure out how to render them as spheres in order to visually validate our progress. Here is an example of what an early testing set up would have looked like:



Although this set up looks like a cloth, the springs themselves are able only to bend and the particles will repulse each other to simulate incompressibility.

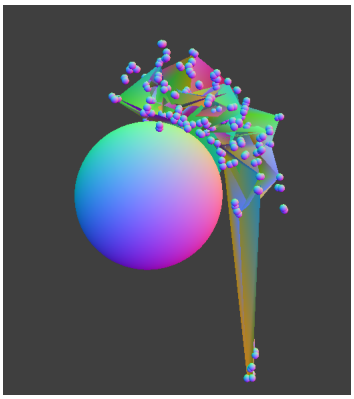
3.3.2 Particle Simulation Tests. After we had the basic cloth like set up, we needed to test the ability of the particles not just to link together, but to *move* together like water. To this

end we made use of our container and sphere primitives to see how the falling and moving particles would react when interacting with other objects at varying gravities. One of those tests might have looked like the following:



Notice how the particles scatter and redistribute themselves as they hit the sphere. This testing framework allow aided us in ensuring that our incompressibility functions were properly working.

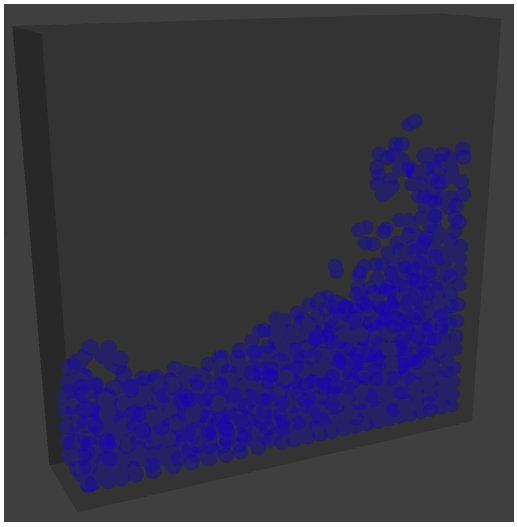
3.3.3 Spring Tests. The final component of our testing revolved around making sure that the springs would properly slow and eventually halt the water as it froze. This section of testing largely consisted of running the simulation, pausing it, setting the parameters to freeze the water, then allowing the simulation to continue. This allowed us to visually ensure that the simulation would react more rigidly the "colder" it was. Here is an example of what that might look like:



This image is rendered with normal shading as that best displays the how the springs connect to each other. This is an image of water being frozen immediately after splatting onto a sphere.

4 RESULTS

In the end, we were able to render fluids which moved and behaved generally realistically. Although it is difficult to show the results without video, here is a frame taken from one of our final renders.



This render began with the water particles bunched up in a rectangle at the top of the container. They then fall and splash around the container until settling.

The freezing part of the simulation was slightly less of a success however. Our initial spring based implementation did not work as intended, so we instead opted to use correction vectors to simulate freezing. Although this does work in slowing the simulation, it can sometimes result in a final product that is gittery.

5 PROBLEMS ENCOUNTERED AND LESSONS LEARNED

Throughout this project, a number of the things we initially thought would be quite easy actually ended up becoming huge issues. For the sake of posterity (and even humour) here are a number of the largest issues we faced.

5.0.1 Initial Renders. It was actually quite a challenge for us to initially render the particles in our system. Our original approach was to create a new class, called a particle, and use that in place of the point masses in the starter code. The idea was to embed a point mass inside of a particle and just access it whenever needed. This ended up causing unending issues. In particular, the code did not seem to like it when we changed the types of vectors from point masses into particles. Nothing would render, even though we confirmed over and over again that our approach was correct. Eventually we worked around this issue by simply converting everything back into point masses as in the base code and using getting functions to render them in the cloth simulator class. The moral of the story is a simple one: it is better to build on top of a code base which you do not fully understand than to try an change it fundamentally. You never know what little change will unravel the whole thing. Another important lesson here is that sometimes it is better to abandon a strategy and start

over on a new path than to continue to try to fix something broken. We lost a lot of time on the project before we even implemented most of it just because of this rendering issue.

5.0.2 Kernel Specifics. The paper we based much of this simulation on is relatively sparse in details. Perhaps it is just because our group is generally inexperienced in reading technical papers, but we ended up having to do a lot of additional research and ask many questions of the professor in order to properly implement our kernel functions. In fact, we ended up using an entire additional research paper to supplement the functions in the first one because they were not detailed enough. Our lesson here is that technical papers are writing for technical people, people who do not need things to be spelled out for them in nearly as much depth as we do. This meant that we actually had to put a lot more research into the math and particular equations we were implemented than we originally intended.

5.0.3 Picking Constants and Exploding Particles. Naturally we expected a degree of number twiddling when implementing this project, but we had no idea just how much work it would actually be to find constants that not only work but look good in our simulation. Much of our work in the final days boiled down to testing a variety of different constants: the rest density, kernel length, and particle radius, etc. This compounded with small mistakes to create a nightmare of debugging.

As a salient example, we initially miscalculated the neighbors of each particle due to a bad spatial implementation, resulting in particles not properly enforcing incompressibility. We didn't flip the negative sign when computing the gradient with respect to the particle's position. We fixed all of this only to then realize that our constants were incorrect as well.

Even worse is that often times it was difficult to distinguish between code that was buggy and code that simply did not have the right constant. Early on we had an issue where all of our particles would seem to just explode outward whenever the simulation started. Although it presented an oddly beautiful sight, evocative of the night sky on a clear moonless night, it was not what we intended to happen. We spend huge amounts of time trying to fix our kernel functions and rewrite our simulation loop, only to eventually find that every thing was fine, it was just the a constant we selected in one of our kernel functions was blowing up and causing the explosion.

5.0.4 (Tw)ice (Tw)ice Baby: our Second Spring System. Reading this paper, you it may have struck you as somewhat odd that we used a mass and spring system in the project without ultimately using it to apply forces. This is because we originally tried to do this, but the initial spring system we implemented did not work particularly well. As we changed our spring constant, it would cause particles to approach each other in way that looked unnatural and also violated incompressibility. Ultimately we scrapped the idea of using spring forces to simulate freezing, instead using correction vectors to do it. We determined it useful to leave the original spring system in as a structural component, since it already did the work of describing the neighbors, and just add in the correction vectors. In hindsight, perhaps the use of a spring system was not the best way to simulate ice movement. Much of what we did in the simulation could probably have been accomplished by clever use of damping functions instead. It goes to show that sometimes the cool thing you learned about in class is actually a lot more complicated than what you need to make something work.

5.0.5 Texturing. Texturing our particles turned out to be quite difficult. We found that we were able to texturize each individual particle, but doing this at the level of single particles made the final rendering unrealistic. We then attempted to add particles in between the main particles so that the entity looked more uniform, but we found that this slowed the simulation dramatically and still proved unsatisfactory. We then attempted to solve the issue with a mesh rather than a simple texture by creating one for the water. However, we found that generating the mesh was difficult as identifying the "surface" points proved to be an a not insignificant challenge. Another roadblock that we came across was finding a way for the surface to look "water-like". Since water has a transparent property, and the texture and color of it changes by depth, we found that changing the texture as the water depth change was another challenge. We later found that one workaround for texturizing water was to use unity; we could generate the points and meshes in unity and have unity texture the water for us. Ultimately we discovered that making the final renders look nice was, in many ways, as complex a task as implementing the fluid sim in the first place. As as result our final renders suffered to some degree.

6 FUTURE WORK

This project leaves several avenues open for extension. One of the more obvious of these is simulating other state transitions. In particular, the water simulation could be extended to turn into steam rather than ice. This could also involve a spring system, which instead of causing particles to stick together, would push them apart.

The project currently does not make full use of the spring system as is described in the issues and lessons section above. We do not use it to calculate and account for spring forces, so a future project could consider incorporating that as well as other forces into the simulation.

A current limitation of our project and something that would also be relevant to a larger project is the expansion and contraction of particle sizes. Although our simulation turns water into ice, it does not actually enlarge the size of the particles as they freeze, something one might expect from an ice simulation. Implementing a feature like this could also be useful in implementing a steam simulation as the steam would need to simulate spreading out into the air as it rose up from the water.

Another limitation of our project is that it freezes all particles of water at the same rate. There is potential to implement a system which instead freezes water in different parts of the simulation at different speeds. This would involve a large degree of randomization and would require the springs to store far more variables as they would be responsible for managing their forces rather than the simulation as a whole.

7 CONTRIBUTIONS

Alex Schedel: Design and code structure, rendering implementation, spring system implementation, additional primitives and test scenes, authoring final paper (as the one grad student in this group it only felt fair)

Lucy Lu: Implemented and debugged spatial mapping, fluid simulation, spring distance correction, and state transition. Refactored codebase to match simulation loop implementation in paper. Determined constants, such as rest density and kernel length. Implemented container and collision code. Implemented GUI panel and opacity change as temperature changes.

Olivia Huang: Implemented incompressibility position correction vector calculation and researched gradient functions. Rendered videos for various container sizes and situations.

Eddy Byun: Implemented the spiky gradient kernel and researched and attempted to render the particles with texturing.

REFERENCES

[1] ADITHYA VIJAYKUMAR. 2012. Smoothed Particle Hydrodynamics Simulation for Continuous Casting.

[2] Matthias Müller, David Charypar and Markus Gross. 2003. Particle-Based Fluid Simulation for Interactive Applications. https://matthias-research.github.io/pages/publications/sca03.pdf. [Online; accessed 5-April-2023].

[3] Miles Macklin and Matthias Muller. 2013. Position Based Fluids. https://mmacklin.com/pbf_sig_preprint.pdf. [Online; accessed 5-April-2023].

[4] Ren Ng and James O'Brien. 2023. Fluid Simulation.