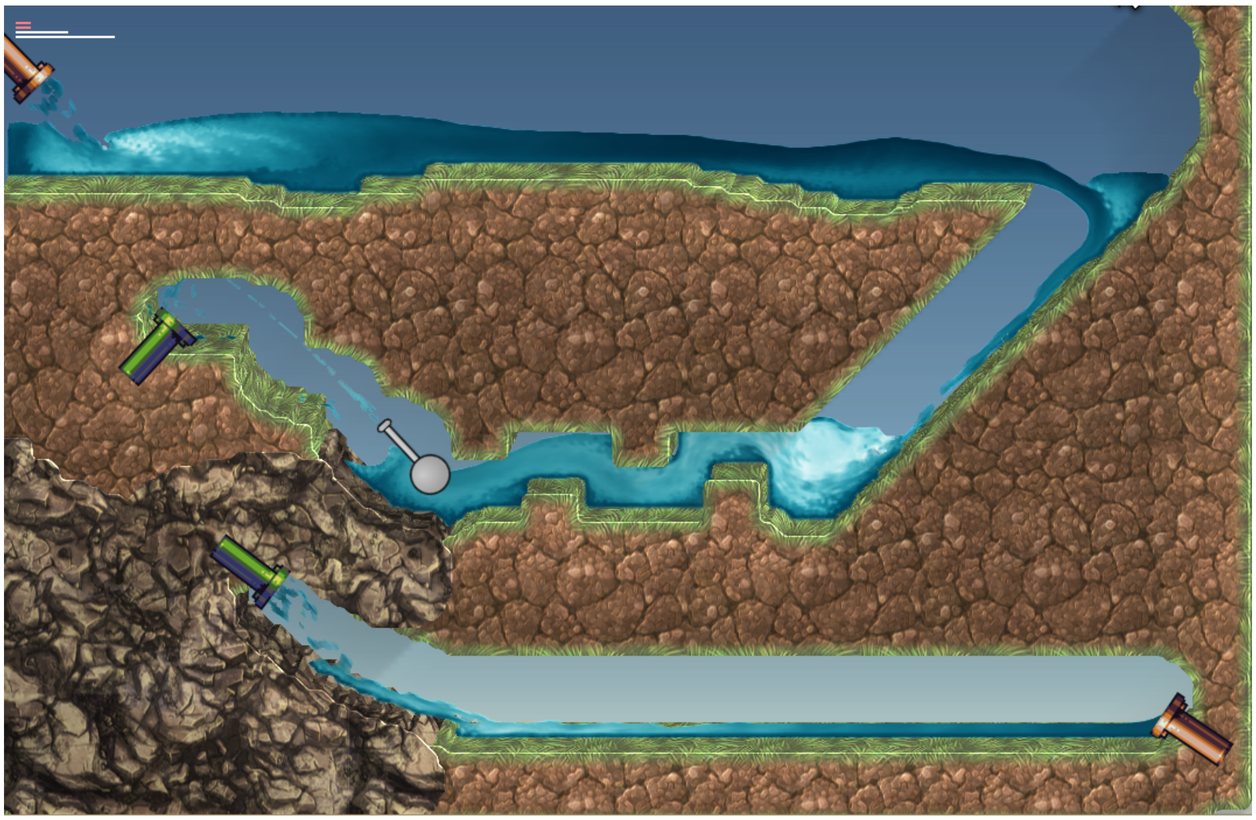
BRING BACK WINTER!

Interim Report



Current Status

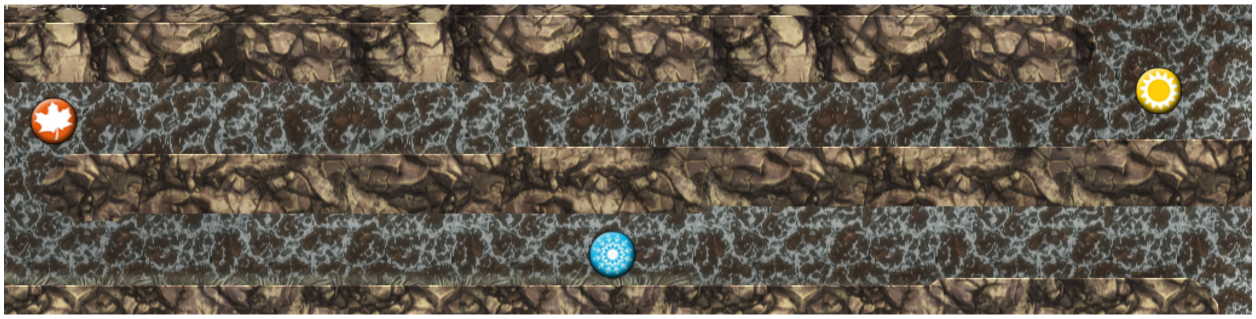
Both the functional minimum and the low target are fully implemented. In addition, the majority of the desirable target and some of the high and extra targets have been completed.

Implemented Features

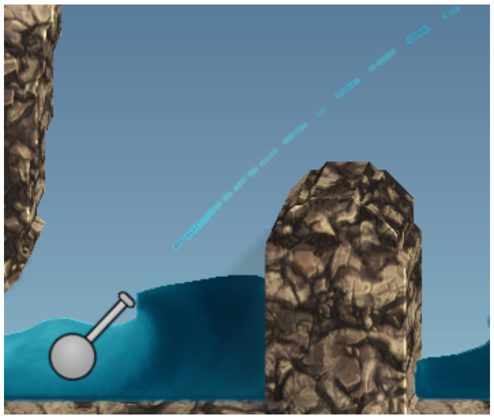
Implemented features	Functional Minimum	● Fluid Simulation ● Editable terrain ● Extremely basic, side view rendering ● Sandbox style gameplay
	Low Target	● Season changers ● Switches and gates ● Season sensitive gates
	Desirable Target	● Water pumps ● Water pipes ● Textured terrain
	High Target	● Music / Audio
	Extra Target	● Post process refraction shader
Features in progress	Desirable Target	● Season dependent graphical effects

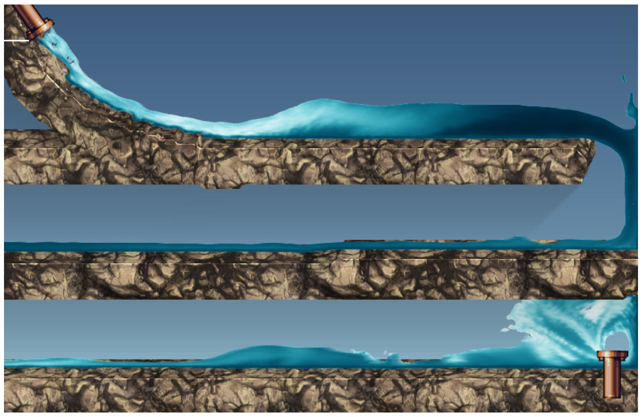
Gameplay elements

Season changers

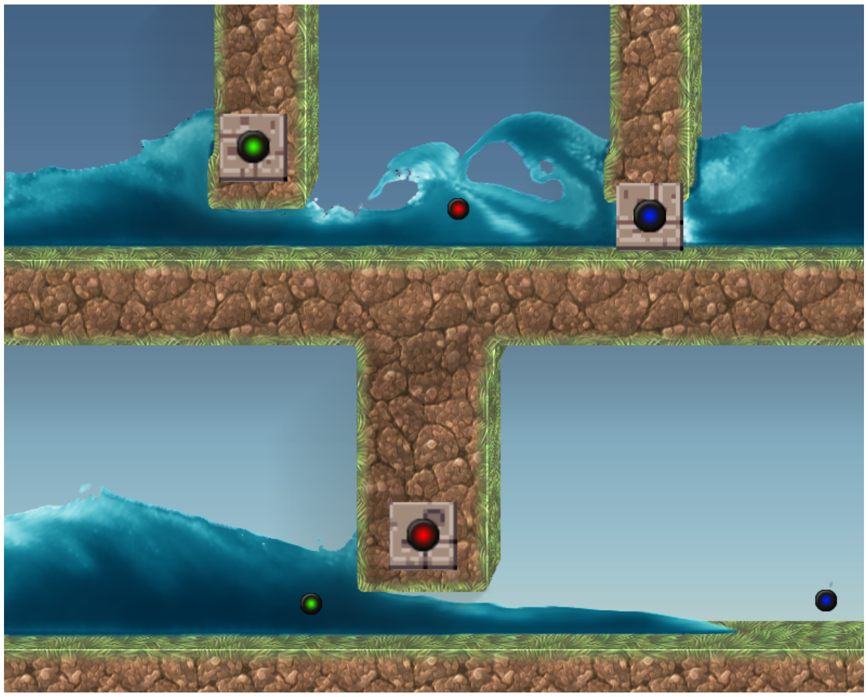


Pumps & Pipes





Gates



Technical challenges

The core gameplay of “Bring Back Winter” revolves around a 2D fluid simulation, and thus our first priority was to implement a fluid solver and make it fast and robust for real-time use.

We chose Smoothed Particle Hydrodynamics (SPH) as the basis of our implementation. Although SPH can be inferior in terms of preserving turbulent flow, it can be made real-time at relatively affordable cost compared to Eulerian simulations. It requires little memory and easily handles free surfaces flows, which are important to our gameplay.

Implementing SPH in a robust and fast manner turned out to be challenging. SPH comes in many flavours, and most of them restrict the time step to preserve robustness and convergence. To make it real-time however, we needed to run the simulation at 10-20 times the recommended timestep. This created a plethora of numerical issues, which regularly caused oscillations and caused the simulation to blow up frequently.
We were able to fix these issues by consulting literature and adding various non-physical damping factors and clamps. The fluid is far from being physically correct, but it still “feels right”, preserves interesting fluid flow and is robust even under extreme conditions. The only remaining issue is that particle near boundaries tend to oscillate slightly under high pressure, which is more of a visual nuisance. This was fixed with a few rendering tweaks that hide this problem from the user.

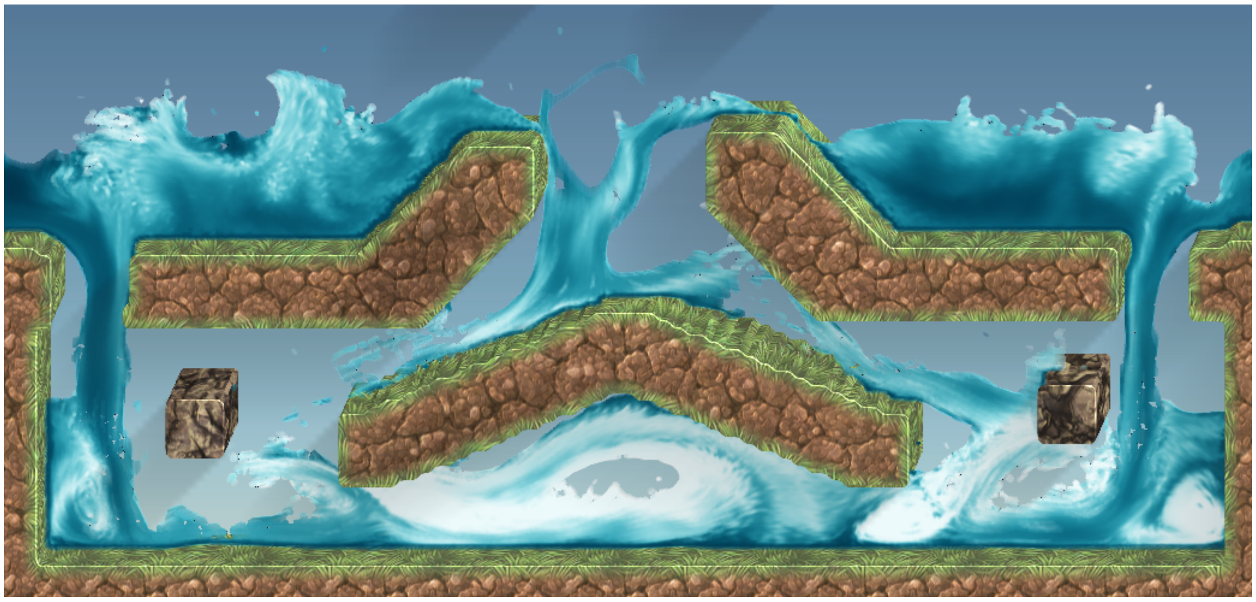
Performance was also an issue. A straightforward SPH implementation makes use of a 2D grid and a linked list of particles in each grid cell to accelerate nearest neighbour queries. However, with many particles this thrashes the cache and makes the code memory bound for large numbers of particles. We were only able to simulate up to 5’000 particles at 60FPS, even with fairly optimized numerical code.

To solve this problem, particles are now sorted after each iteration using a linear time counting sort. Particles are rearranged based on the grid cell they belong to. This gets rid of the linked list and places particles that are near to each other spatially close together in memory. The overhead of the sorting step is negligible, but the new particle layout is cache friendly and resolves the thrashing issues. Due to the large number of computations and cache friendly behaviour, the code now becomes compute bound and is stuck simulating at most 7’000 particles at 60FPS.

This was not enough for our taste, so we improved our code to make use of vector instructions. During the sorting step, the particle attributes are now rearranged into a 64 byte aligned, struct-of-arrays layout and are padded such that the number of particles in each grid cell is divisible by the SIMD vector length. This way, all particle-particle interactions can be fully vectorized such that one particle computes interactions with up to 8 other particles in parallel.

We tested both SSE and AVX instructions and found performance to peak with SSE at nearly 20’000 particles at 60FPS. We restrict the used intrinsics to SSE2, which are available in nearly every processor nowadays.

As a last step, we also added multithreading. The sorted list of particles is divided up between multiple threads that synchronize after each simulation step. With 4 threads, this enables another considerable speedup and allows us to simulate 60’000 particles at 60FPS. This is way more particles we will ever need for the game, and the performance goal is met.



Rendering

Rendering the fluid also turned out to be difficult. For particle based simulation methods, “blobby objects” such as metaballs are usually chosen, and they are fairly convenient to implement in 2D. However, classic metaballs don’t adapt to the local particle distribution and both hide interesting features in the fluid flow and also cause a terribly “bumpy” looking water surface.

To resolve these issues, we consulted literature and implemented several papers. To fix the bumpy artifacts, we now use anisotropic metaball kernels that adapt to the local fluid flow based on the variance in the particle distribution. To better show the internal features of the fluid, we implemented an aeration model that computes the water-air mixture inside the fluid. At splashes on the water surface, air is mixed into the water and is transported and diffused in the fluid flow. Based on the fraction of air transported with the fluid, we locally modulate the absorption coefficient in the water to generate nice exponential falloff curves.

As a bonus, we also added post process motion blur to the water to emphasize interesting flow.