

Report

<https://www.dropbox.com/s/xl9jpr9hfay8l3/mini-project%20-%20Google%20Chrome%202021-12-16%2017-59-12.mp4?dl=0>

Criterion 4

Wind (horizontal acceleration)

Rearranging $F=ma$ to get acceleration = F/m , and simplifying by calling the mass of a particle 1 (arbitrary mass unit), we get acceleration = force. By altering the acceleration slider, the wind, which acts horizontally on the particles that are above the sides of the pot, causes the given acceleration – this is implemented by adding the value of the acceleration to each particle's velocity every time they're redrawn.

Temperature (vertical initial velocity)

I spent much longer on this property of motion – I set the initial velocities (vertical) of the particles as they evaporate based on Maxwell-Boltzmann distributions and probabilities of particles having an energy corresponding to certain temperatures given the temperature of the water [1].

For any large number of molecules in thermal equilibrium at the temperature T , the probability P that a given molecule has energy lying between E and $E + dE$ is given by

$$P = Ce^{-\frac{E}{kT}} = Ce^{-\frac{mv^2}{2kT}} \quad (4.7)$$

where the coefficient $C = (m/2\pi kT)^{3/2} 4\pi v^2$ varies with T but at negligible rate compared with the exponential factor $\exp(-E/kT)$. The exponential factor gives the fraction of the most energetic molecules whose energy exceeds a given level E that might be arbitrarily chosen, e.g. a specific energy threshold.

Using the equation above, you can calculate the probability that a molecule has over a certain energy (therefore can calculate the probability of a particle having less than a certain energy by subtracting it from 1, and use that to calculate probability ranges).

You can use the energy in the equations $E=(3/2)kT$ and $E=(1/2)mv^2$ to calculate the velocity corresponding to certain energy (therefore temperature) ranges (see image below for maths).

```

sketch.js | particle.js | emitter-group.js | emitter.js | index.html
}

setTemp(temp) {
  if (this.temp != temp) {
    // Calculate probabilities of initial velocities
    this.probabilityCumSum = 0;
    for (var i = 0; i < this.noOfBins; i++) {
      // midpoint value of the bin, e.g. for temp bin 0-10 deg. C, binTemp = 5
      this.binTemps[i] = this.lowestTemp+(i*this.sizeOfOneBin)+(this.sizeOfOneBin/2);

      // Calc probability that particle has energy corresponding to each temperature bin
      if (i==0) {
        this.particleTempProb[i] = 1-Math.exp((-1.5*(this.binTemps[i]))/(temp));
      } else {
        this.particleTempProb[i] = 1-Math.exp((-1.5*(this.binTemps[i]))/(temp)) - this.probabilityCumSum;
      }
      this.probabilityCumSum += this.particleTempProb[i];


      // If the energy that a particle has corresponds to <100 deg. C, it won't evaporate, so set velocity = 0
      if (this.binTemps[i]<100) {
        this.particleInitVelocities[i] = 0;
      } else {
        this.particleInitVelocities[i] = this.getVFromT(this.binTemps[i]);
      }
    }

    this.temp = temp;
  }
}

```

Each temperature bin has an associated velocity with it that remains the same regardless of the water temperature; the probabilities of a particle being in a particular temperature bin change as the water temperature changes. Particles with energy corresponding to less than 100 degrees Celsius have their velocities set to 0. It is noted that the physics I have used is extremely naïve and should use Kelvin instead of Celsius, however due to the technicality that many more particles than desired for the intended visual effect would have the right energy to evaporate when using Kelvin, I used Celsius instead.

Temperature: 50°C



Probability of a particle having energy corresponding to a temperature within a range, and the corresponding velocity of emitted steam particles in each bin (ranges < 100°C mean the particle doesn't have enough energy to evaporate therefore velocity = 0 pixels/s):

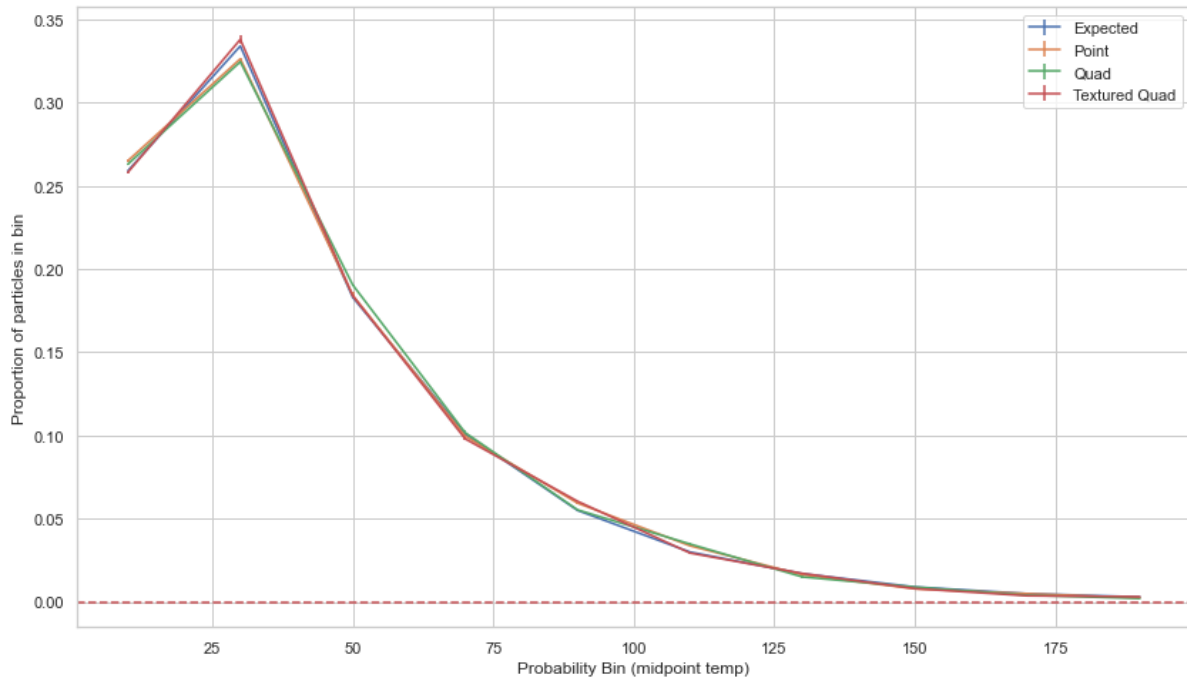
P (0°C - 20°C) = 0.259	→ 0 pixels/s
P (20°C - 40°C) = 0.334	→ 0 pixels/s
P (40°C - 60°C) = 0.183	→ 0 pixels/s
P (60°C - 80°C) = 0.101	→ 0 pixels/s
P (80°C - 100°C) = 0.055	→ 0 pixels/s
P (100°C - 120°C) = 0.03	→ 1.919 pixels/s
P (120°C - 140°C) = 0.017	→ 1.969 pixels/s
P (140°C - 160°C) = 0.009	→ 2.017 pixels/s
P (160°C - 180°C) = 0.005	→ 2.064 pixels/s
P (180°C - 200°C) = 0.003	→ 2.11 pixels/s

The top slider represents the temperature of the water, and underneath are the probabilities of emitted particles at the current water temperature having energy corresponding to different temperature ranges, and the values after ‘->’ show the corresponding speed of a particle in that bin (these speeds don’t change).

To demonstrate that my program follows the probability distribution I have defined, I made a function 'getObservedBinProbability()' that will print the actual proportions of on-screen particles that are in each temperature bin, and I experimented by collecting 3 samples (one every 5 seconds) with each rendering type, showing that the actual distributions are always very close to the expected distribution (on the left).



Example run of the function in the console



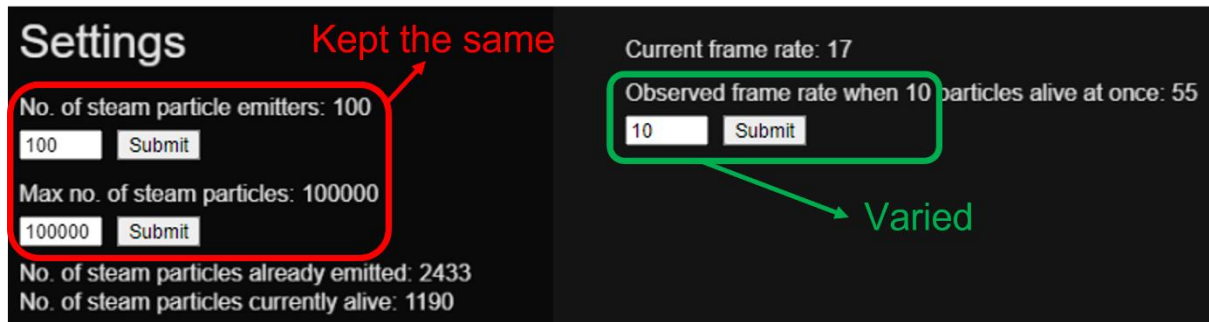
The overlapping lines show that my program works as expected from the laws I defined.

Criterion 5

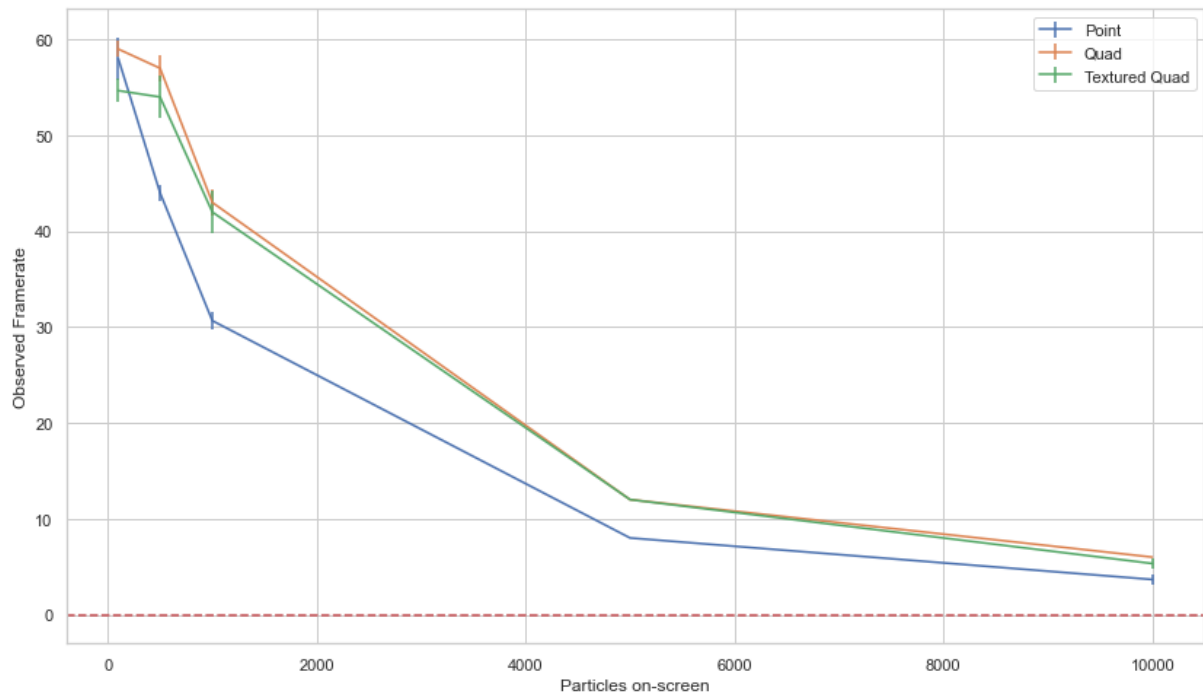
To make the program more efficient, instead of splicing the particle array to remove dead particles (which would be costly performance-wise because not only would it remove the particle, it would need to reduce the array size and move the particles after the removed particle down the array), I use 'delete' to set the array element undefined but not reindex/update array length. Then after every 5 seconds, I call a function to clean the particle array by removing the undefined elements in the array and resizing it down. This prevents the array from becoming very large since particles are being pushed onto the array when they are emitted, so if we never clear out the undefined elements the array will keep on growing and be mostly full of 'undefined', taking up memory unnecessarily. I chose multiple seconds delay for cleaning the array instead of constantly clearing it on each iteration as a trade-off; the more memory you can spare, the longer you could set the time between array cleaning, and the longer the time is between cleaning, the better the performance will be because the cleaning takes time to do so needing to do it less often gives better visual performance.

Criterion 6

The way that I designed the program such that particles would be generated based on probabilities didn't allow me to logically set how many particles were on-screen manually, so to work around this I conducted an experiment where I captured the observed frame rate when 100, 500, 1000, 5000, 10000 particles were on-screen at once (my laptop can't handle much more unfortunately, it's very old!) by setting the maximum time to live and temperature and letting it run until the desired number of particles was reached.



Graph of results after 3 iterations per setting, and for each rendering type:



In general, as the number of particles on-screen increases (no. of particles in memory at a given time), the observed frame rate (therefore the performance) decreases and the shape of the graph shows exponential decay.

There are consistent differences in the performances of the different rendering types in that p5js Point is consistently the worst performing, apart from the first case with 100 on-screen particles but the overlap of the error bars of the 3 lines shows that the 3 rendering types are not starkly different performance-wise at 100 particles. To render the quads, I used p5js 'square' and interestingly the performance is similar for textured vs. non-textured quads, the textured quads only slightly worse performing than the non-textured quads.

A bottleneck in my design is the cleaning of the particle array because to clean it, a filter is applied to find all the undefined elements (which will have to examine each element of the array $O(n)$), and then remove these elements, resize the array, and move the elements down that are not undefined. In relative terms it is quite a heavy process especially if done frequently, so to mitigate it I made it only execute every 5 seconds, but to optimise this you would conduct experiments to find the best trade-off between how much memory on your particular system is ok to use before the array MUST be cleaned, thus minimising the frequency of the cleaning. You would do this by varying the time between cleans, and capturing observed framerates.

References

[1] Camuffo D. Consequences of the Maxwell–Boltzmann Distribution. In: Camuffo D, editor. *Microclimate for Cultural Heritage (Third Edition)*. Amsterdam, Netherlands: Elsevier; 2019. P. 61-71.