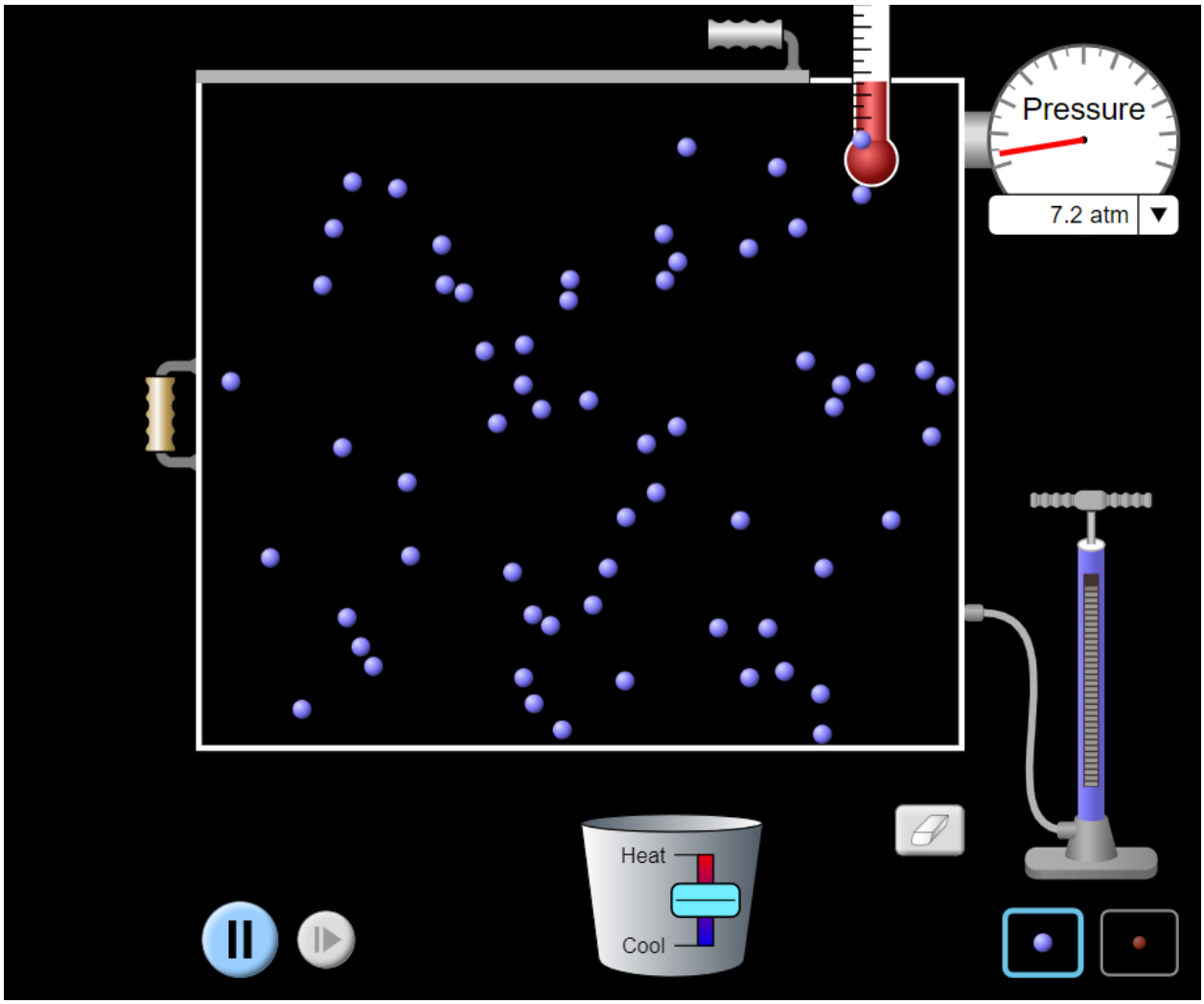
Note: Reading through the documentation for both weeks may be beneficial, because then you can design week 1's code in a way that is easily extensible to week 2.

The goal of this assignment is to simulate an ideal gas and use Cinder to visualize it. It may be helpful to play around with this online simulation to get an idea of what we're trying to build:

https://phet.colorado.edu/sims/html/gas-properties/latest/gas-properties_en.html



Week 1

Our simulation will consist of a bunch of identical circular particles elastically bouncing around in a (two-dimensional) box. Every particle will have a position and a velocity; the position and velocity are 2D vectors. (Note that position refers to the *center* of the particle.) Instead of storing the x-component and y-component of each vector in separate variables, we recommend storing them using the **glm::vec2** class (glm is the namespace for the OpenGL Mathematics library). This will make a lot of the computation simpler to code, because we'll be able to use vector operations.

We could initialize some 2D vectors using the following lines of code (this is the syntax for calling the vec2 constructor):

```
#include "cinder/gl/gl.h"

glm::vec2 position(5.5, 6.7);
glm::vec2 velocity(0.4, -0.6);
```

Warning: do NOT use `#include <glm/vec2.hpp>` ***to include the vec2 class. Some students in previous semesters did this and ran into mysterious problems. Just stick to*** `#include "cinder/gl/gl.h"` ***as shown in the example above.***

On every frame of our Cinder application (i.e. each time the **update** function is called), we'd like to advance our simulation by one unit of time. The FPS of the Cinder application is irrelevant; we'll just assume one unit of time has passed every time the update function is called.

Change in position = velocity * time, so after one unit of time, the position vector should be updated to `[5.9, 6.1]`. This can be achieved with the following line of code:

```
position += velocity;
```

Note that this is possible because the vec2 class has overloaded the += operator. (If we stored the x and y-components separately, this would require two lines of code instead.)

At this point, we know how to make a single particle move. However, if we never change the velocity, the particle will just move in a straight line forever. A particle's velocity needs to be updated whenever it collides with a wall or with another particle.

When a particle collides with a vertical wall, the x-component of the particle's velocity should be negated. Similarly, for horizontal walls, the y-component should be negated.

For collisions with another particle, the math is a bit more complicated. A collision will occur if the distance between the two particles' centers is less than or equal to the sum of the two particles' radii (i.e. the two particles are touching). After Particle 1 collides with Particle 2, the new velocity of the particles can be calculated with these equations:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{(\mathbf{v}_1 - \mathbf{v}_2) \cdot (\mathbf{x}_1 - \mathbf{x}_2)}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2)$$

$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{(\mathbf{v}_2 - \mathbf{v}_1) \cdot (\mathbf{x}_2 - \mathbf{x}_1)}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

In the equations above, $\mathbf{v}_i$ is the 2D vector representing the original velocity of Particle i, and $\mathbf{x}_i$ is the 2D vector representing the position of Particle i. $\mathbf{v}_i'$ represents the new velocity of Particle i after the collision. **a** · **b** represents the dot product of two vectors, and **||a||** represents the length of a vector. As a reminder, the glm library can be used to perform various vector operations (including dot product and length); we'll leave it up to you to figure out how.

(If you are interested in the derivation of this equation, this may be a helpful resource: https://en.wikipedia.org/wiki/Elastic_collision#Two-dimensional.)

As an example, if our simulation has two particles, each with radius 1:

- Particle 1: position = [19.9, 20], velocity = [0.1, 0]
- Particle 2: position = [21.5, 21.4], velocity = [-0.1, 0]

After one frame, their positions will change:

- Particle 1: position = [20, 20], velocity = [0.1, 0]
- Particle 2: position = [21.4, 21.4], velocity = [-0.1, 0]

Now, the distance between the particles is approximately 1.98, which is less than the sum of their radii (1 + 1 = 2), so the two particles have collided, and we need to update their velocities. Using the equations above, we can calculate the new velocity of Particle 1:

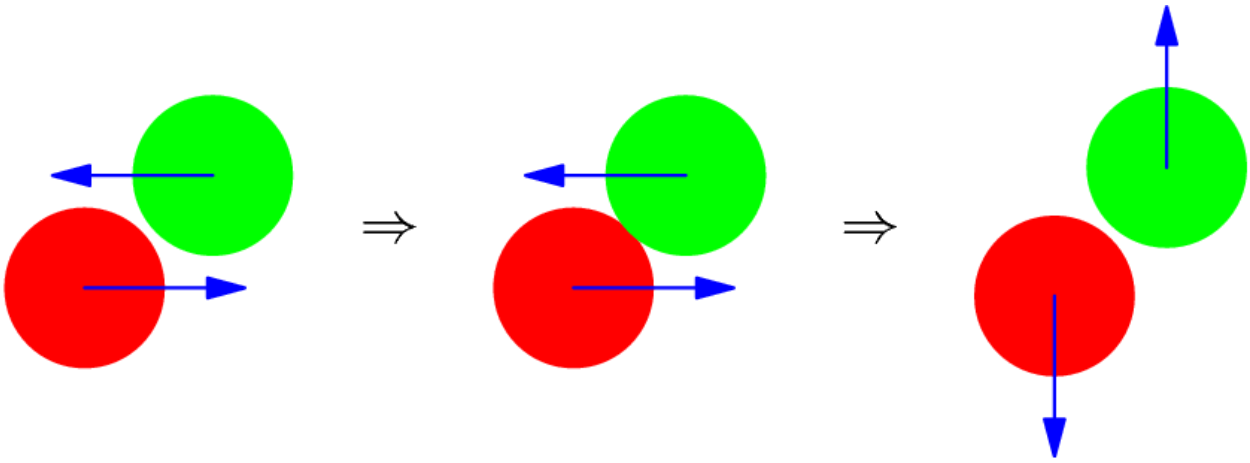
$$\mathbf{v}_1 - \mathbf{v}_2 = [0.1, 0] - [-0.1, 0] = [0.2, 0]$$
$$\mathbf{x}_1 - \mathbf{x}_2 = [20, 20] - [21.4, 21.4] = [-1.4, -1.4]$$

$$\begin{aligned}
\mathbf{v}_1' &= [0.1, 0] - \frac{[0.2, 0] \cdot [-1.4, -1.4]}{\|[-1.4, -1.4]\|^2} [-1.4, -1.4] \\
&= [0.1, 0] - \frac{-0.28}{3.92} [-1.4, -1.4] \\
&= [0.1, 0] - [0.1, 0.1] \\
&= [0, -0.1]
\end{aligned}$$

The calculation for Particle 2's new velocity is similar. You should be comfortable with performing these calculations so that you're able to cross-check results in your test cases. After updating the velocities, we also need to update their positions. So after two frames, the state of our simulation is:

- Particle 1: position = [20, 19.9], velocity = [0, -0.1]
- Particle 2: position = [21.4, 21.5], velocity = [0, 0.1]

Here's a diagram illustrating the three frames in this example:



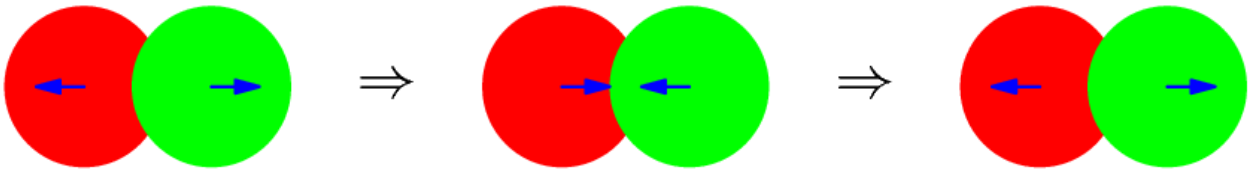
Finally, inside the **draw** function, you should draw all of the particles, along with the rectangular boundary of the box.

It's up to you to determine parameters like the number of particles, radius of particles, size of the box, and the initial position/velocity of the particles. Fiddle with the parameters to make a simulation that looks cool! (Some configurations will be pretty boring, e.g. if the velocities are too slow or if collisions rarely occur because there aren't enough particles. Thus, it's important to avoid magic numbers so that your program is flexible/easy to change.)

Some additional things to be aware of:

When setting your initial velocities, make sure they are relatively small compared to the radii of your particles. This is necessary because in the simulation, speedy particles might be able to "tunnel" through each other within a single frame without ever colliding, when in reality, they actually would have collided. This phenomenon is shown in the diagram below:

Another glitch you may notice is that particles stick to each other or stick to the wall. This is caused by scenarios such as the one below:



In the first frame, the particles are touching, but they're already moving *away* from each other. If we execute a collision between these two particles, their velocities will reverse and they'll start moving towards each other. Then, in the next frame, we'll execute *another* collision between the same two particles, and they'll start moving away from each other again. This cycle will continue indefinitely, so the two particles will never be able to separate.

To fix this issue, we should only execute a collision between two particles when they are moving *towards* each other. Two particles are moving towards each other if the relative velocity of the two particles is in the opposite direction of their relative displacement. This condition can be mathematically expressed as

$$(\mathbf{v}_1 - \mathbf{v}_2) \cdot (\mathbf{x}_1 - \mathbf{x}_2) < 0$$

The same idea applies to wall collisions; we should only execute a collision between a particle and a wall if the particle is moving towards the wall.

Testing

Remember that you should be testing every piece of functionality and logic in your simulation; just watching the simulation is not sufficient verification that everything has been implemented correctly (and it's not automated testing). However, you don't have to test the GUI with Catch (e.g. you don't have to test that every pixel on the screen is correct), because there's no simple way to do that.

For example, earlier we mentioned that we should only execute a collision between two particles when they are moving towards each other. *This is something that you should be testing!*

Tip: similar to JSON/Mineopoly, you should test scenarios that you can also verify by hand. In Mineopoly, this meant creating smaller boards. In the JSON assignment, that meant using smaller datasets. However, the test inputs still need to be large enough to properly test all of the desired behavior.

Also, note that you won't be able to test any code inside your app class (the class that inherits from ci::app::App). Thus, this class should not do any of the heavy lifting - which means it should mainly contain code that "connects" the UI to your source code.

Extra Credit:

- Allow the user to speed up/slow down the simulation by using keyboard or mouse input
- Load in a simulation configuration from a JSON file that specifies the number of particles, radius, color, etc.
    - You'll probably want to find a C++ JSON library to help you with this
- Allow a user to save a snapshot of the simulation's state to a file, and load the snapshot from that file back into the simulation at a later time
- Support non-rectangular container shapes
    - For example, you could have a hexagonal container
- It's likely that you have an $O(n^2)$ algorithm to check for collisions. Can you improve the efficiency of this algorithm?
    - Idea #1: If you sort all of the particles by their x-coordinates, then each particle only has to check for collisions with other particles whose x-coordinates are nearby.
    - Idea #2: Use a k-d tree, which allows you to efficiently find the points near some query point.
        - You'll probably want to use a library to help you with this
    - Maybe you can come up with a better idea! These aren't the only valid ones.

Week 2

Before you start implementing new features, make sure you improve your code from last week based on feedback from your moderator.

The first new feature we'll be adding this week is modifying your simulation to support non-identical particles. In particular, you should be able to add particles with arbitrary masses, radii, and colors to your simulation. For example, you should be able to have one particle with mass = 1, radius = 1, color = blue, and another particle with mass = 4, radius = 1.5, color = red, within the *same* simulation.

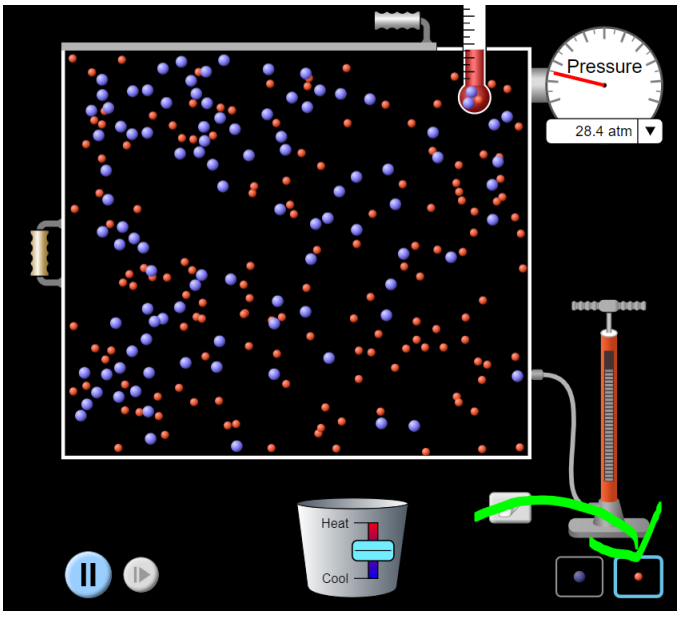
In order to support different masses, the collision equation will have to be changed slightly:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{(\mathbf{v}_1 - \mathbf{v}_2) \cdot (\mathbf{x}_1 - \mathbf{x}_2)}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2)$$

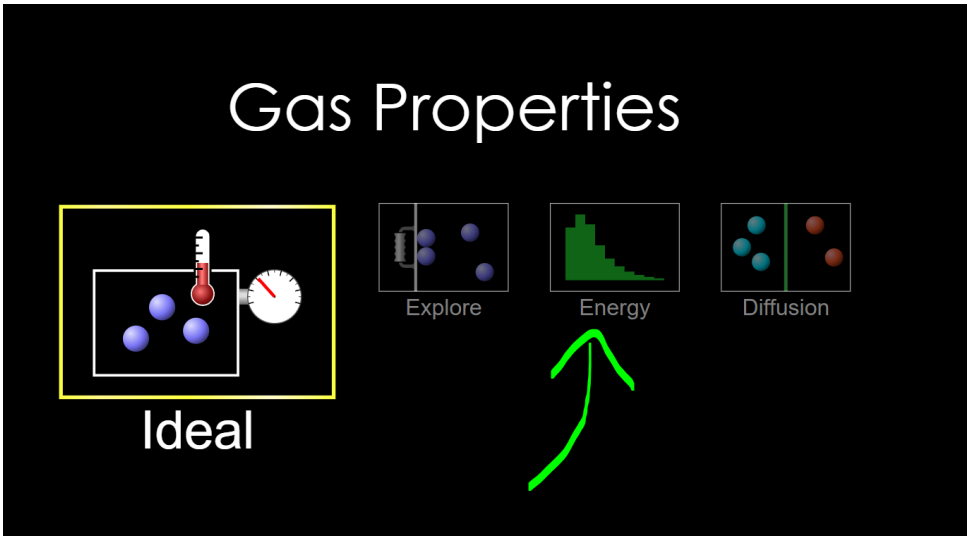
$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{(\mathbf{v}_2 - \mathbf{v}_1) \cdot (\mathbf{x}_2 - \mathbf{x}_1)}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

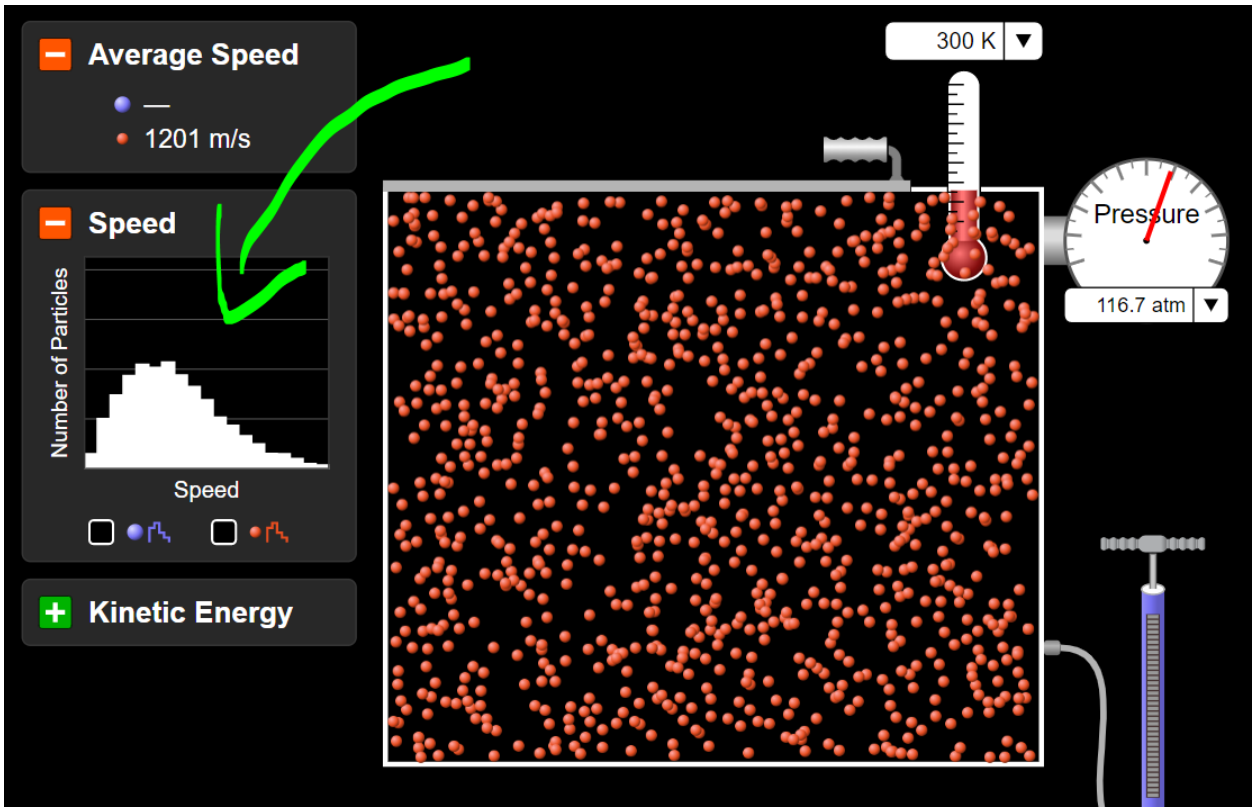
Note that when $m_1 = m_2$, the term with the masses reduces to 1, and we get the same equation that we used in week 1.

The online PhET simulation supports two different types of particles; it may be helpful to take a look at that before you begin implementing your own version. The screenshot below shows how to toggle the type of particle being added:



You may notice that the more massive particles tend to move slower than the lighter particles. The next feature that we'll be adding, a live-updating histogram, will help us visualize this effect. The PhET simulation has a histogram that is similar to what we'll be making. You can access this histogram by selecting "Energy" on the homepage of the website.





You should populate the simulation with three different types of particles; each type of particle should have a mass that is significantly different from than the other types' masses. Then, you will be creating three histograms, one for each type of particle. Each histogram will contain the speed distribution of a single type of particle (speed is the magnitude/length of the velocity vector.)

In total, your app window should contain four things: the box with particles, and three histograms. None of these things should overlap; in other words, they should be drawn in different locations on the screen, and the particle box should not occupy the entire app window.

Your histogram should have labels; the x-axis will be labelled with speeds, and the y-axis will be labelled with frequencies (the number of particles that fall in each histogram bin).

Remember to make your histogram's parameters configurable; that way you can fiddle with the parameters to make the graph look better. (For example, if all of the particles ended up in the same bin, that wouldn't be a very useful histogram!) We're not forcing your histogram to conform to a specific appearance or behavior; feel free to exercise a bit of creativity in this part of the assignment!

If your histogram is very jittery, increasing the number of particles in your simulation will help make the histogram's movements smoother. However, increasing the number of particles by too much may increase the computational costs and cause your frame rate to lag. Improving the efficiency (as suggested in week 1 extra credit) may allow you to overcome this, though.

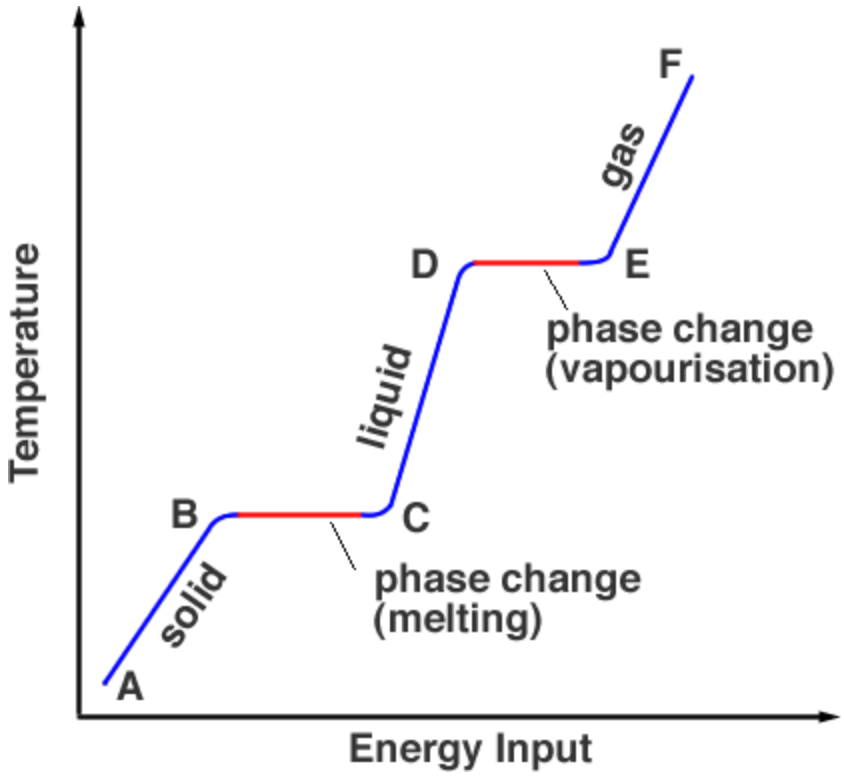
Extra Credit:

- Extra credit from week 1 is still fair game if you didn't implement it in week 1
- Modify your JSON schema to support multiple types of particles
- Overlay the theoretical speed distribution on top of your histogram
- Make the histogram interactive (e.g. add hover effects)

Additional Ideas

If you've enjoyed making this simulation, you may want to consider using it as a starting point for your final project. There are lots of cool ways to extend this; here are some ideas:

- Add intermolecular attraction forces to your simulation
    - This may allow you to observe different states of matter
    - You may remember this graph from high school chemistry. Can you recreate this graph with your simulation?



- Add a gravitational force
    - Does this cause the gas to stratify into layers, where the lighter gas is on top?
    - Can you visualize this by graphing the center of mass of each type of particle over time?
- Create multi-atom molecules (like H_2O)
    - these molecules would also have rotational motion, and the physics behind particle collisions would be more complex
- Turn this into a pong-like game
- Repurpose it to be an infectious disease simulation. The laws governing the motion of "particles" (aka people) would have to change, obviously.
    - You might draw inspiration from this simulation created by 3Blue1Brown: https://www.youtube.com/watch?v=gxAaO2rsdIs