Assignment: A10 (back to main page)

The 2D Baseline

This assignment merges several building blocks developed in earlier assignments. First, the 2D vector class will be used to upgrade our 1D physics engine from the "air-track" assignments. (Note that in this assignment and those that follow, the name "air-table" will refer to the 2D physics environment, much like an air-hockey table. And instead of cars on the air-track, there will be pucks on the air-table.) Second, the network-based server/client classes will be integrated here to allow multiple users to interact with the 2D physics engine. Third, in a way similar to how the client class was used to model the forces from the cursor strings, we will develop a spring class that models the forces that a spring/damper system applies to a pair of pucks.

Python language topics:

- Nothing new here (this assignment may be where it hits you that there is more to writing code than learning the language).

Problem statement:

The first paragraph above makes a good problem statement.

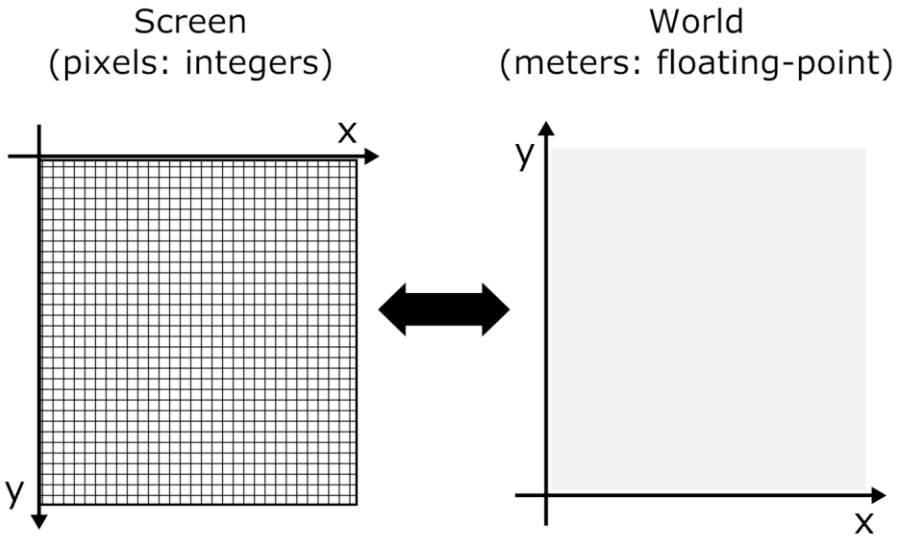
It's best to start with a clean slate here. We will be using concepts learned in previous assignment, but we will not be adding directly to previous code files.

Algorithmic description:

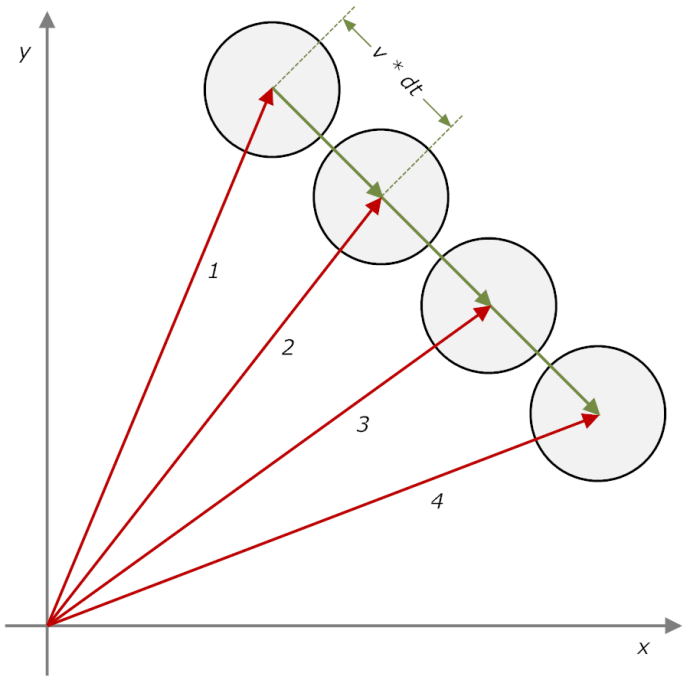
This game loop outline has one or two steps that aren't needed at this point, such as the reference to jets and guns in step 8, but it serves as a good reference for all the assignments that follow.

1. Initialize Pygame.
2. Instantiate the class objects: AirTable, Environment, GameWindow. These act to organize our methods and attributes into conceptual groupings.
3. Start the game's "while" loop and include the following steps:
    a. Check for user input from the local client.
        i. Store keyboard and mouse data in the corresponding "Client" object.
    b. Instantiate all the objects in the game (pucks, springs, controlled pucks). Base this on the local user input that controls the demo selection. Each demo instantiates a different set of objects. This initializes the position and velocity vectors for each puck in the demo.
    c. Check for user input from the network clients.
        i. Store keyboard and mouse data in the corresponding "Client" objects.
    d. Loop through all the clients and calculate the cursor-tether forces (spring and drag) on each user-selected puck.
    e. Loop through all the controlled pucks. Rotate (jets and guns) and calculate the jet and bullet impulse forces on each host puck.
    f. Loop through all the springs and calculate the spring forces (spring and damping) on each spring-connected puck.
    g. Loop through all the pucks:
        i. Sum the vector forces on the puck. Also add in the gravity vector force (m*g).
        ii. Calculate the vector acceleration (acc) caused by the total vector force (Newton's law).
        iii. Calculate the vector change in velocity. dv = acc * dt ; v_final = v_initial + dv.
        iv. Calculate the vector change in position. dx = v_final * dt ; x_final = x_initial + dx.
    h. Check for wall and puck-puck collisions:
        i. Apply penetration corrections to the positions.
        ii. Calculate the post-collision velocities.
    i. Based on the new calculated positions, draw the various objects on the air-table.
    j. Update (flip) the screen.
    k. Increment the time (add dt to the time variable).

It's probably always best to tell a story with pictures. The following drawings and screen shots illustrate algorithmic concepts used in this assignment. Captions are below each image.

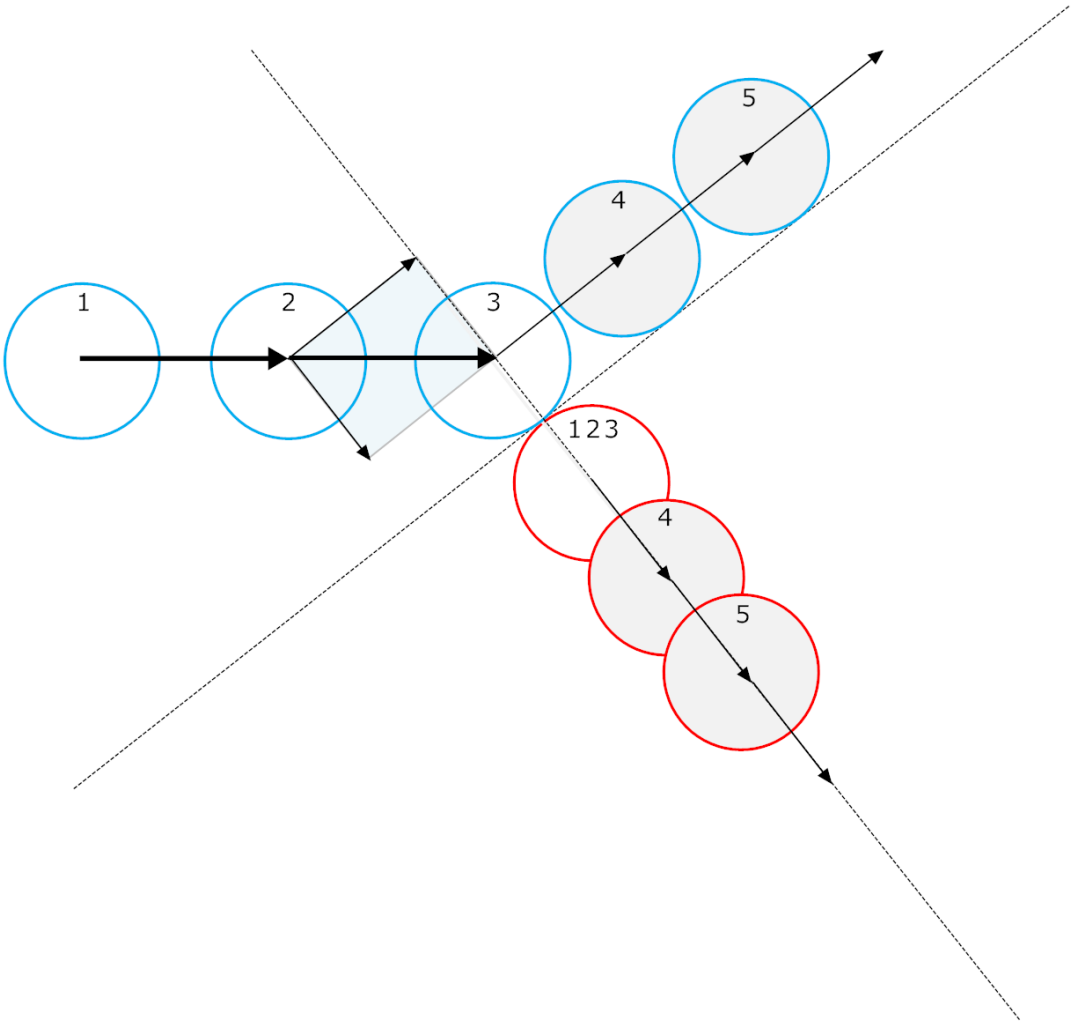


As there was need with the 1D air-track assignments, there will still be need to translate back and forth between the physics world and the Pygame screen in the 2D assignments. With 2D we have to accommodate the inverted orientation of the y axis on the Pygame screen. These transformations can be restricted to one-to-one relationships or they can also support zooming and translation. For starters, try to include a zoom factor in your transformations.



This drawing shows how vectors are used to calculate the changing position of a non-accelerating puck (constant velocity; no external forces). As we did in the air-track assignments, Euler's method is used here to calculate the changes in position. For a more general case, forces must be considered; incremental changes are all based on the external forces applied to objects. Forces cause acceleration (a rate of change of velocity); acceleration causes a change in velocity (a rate of change of position); velocity causes a change in position. But here we use the vector class to account for both components of the calculation; *F*, *a*, *v*, and *x*, are all 2D vectors.

$\vec{a}_n = \vec{F}_n / m$ ➔ $\vec{v}_{n+1} = \vec{v}_n + \vec{a}_n \cdot dt$ ➔ $\vec{x}_{n+1} = \vec{x}_n + \vec{v}_n \cdot dt$

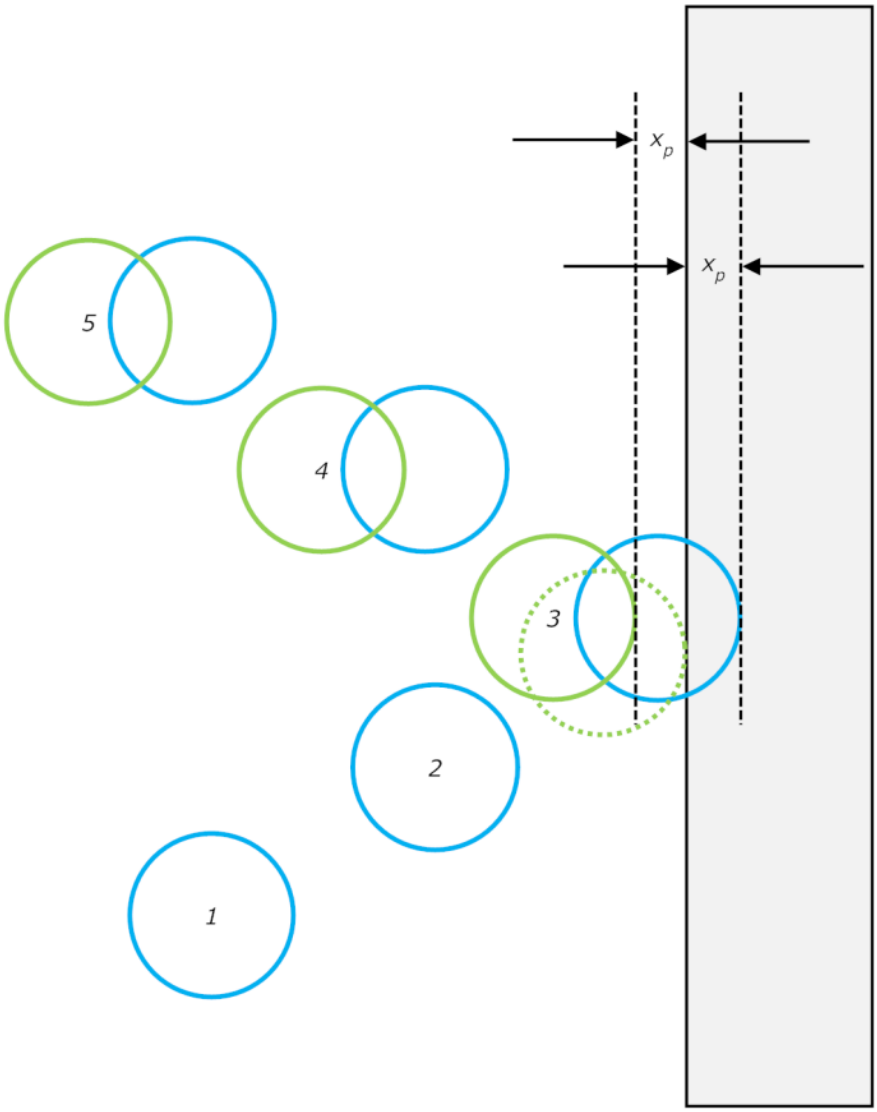


Here the blue puck is approaching (from the left) and collides with a stationary red puck. The sequence is indicated by the numbering of the puck images. (This example is like a typical pool shot where you're trying to get the red ball in the corner pocket.) This shows the axes (dotted lines) of the collision coordinate system, normal and tangential, established based on the contact point of the colliding pucks. One new physics idea is needed to predict the outcome of the 2D collision: that the component of motion of the balls along the tangential direction is unaffected by the collision. This can be seen here in the tangential motion of the blue ball: the tangential component of the approaching ball (image #2) is equal to the tangential component of the ball after the collision (images #4 and #5).

The other physics we need is in the pair of 1D-collision equations from the air-track assignment on car-car collisions. It turns out that the collision physics along the normal direction of the 2D case is identical to the collision physics of the 1D case. Intuitively this makes sense in that we expect the impulse force (the punch) to be delivered along the direction of the contact, the contact normal.

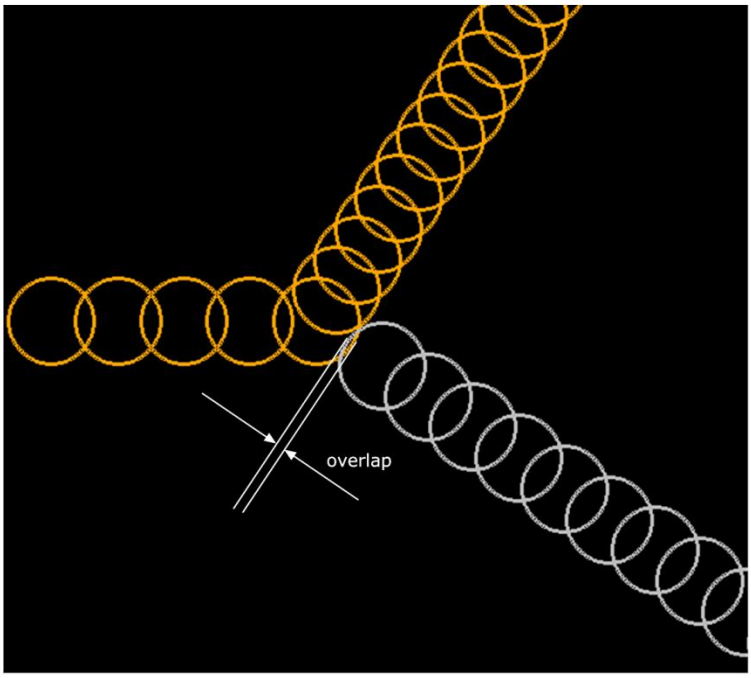
This 2D puck-wall collision drawing is similar to the 1D car-wall drawing that is in the air-track wall-collisions assignment. This shows collision physics ideas similar to those above: just the normal component is affected by the collision; it reverses. The tangential component of velocity is unaffected.

This also illustrates collision detection (the state of overlap or penetration) and the penetration correction that is needed to avoid stickiness behavior. Stickiness manifests when overlapping pucks and objects cannot separate and remain in a constant state of collision.

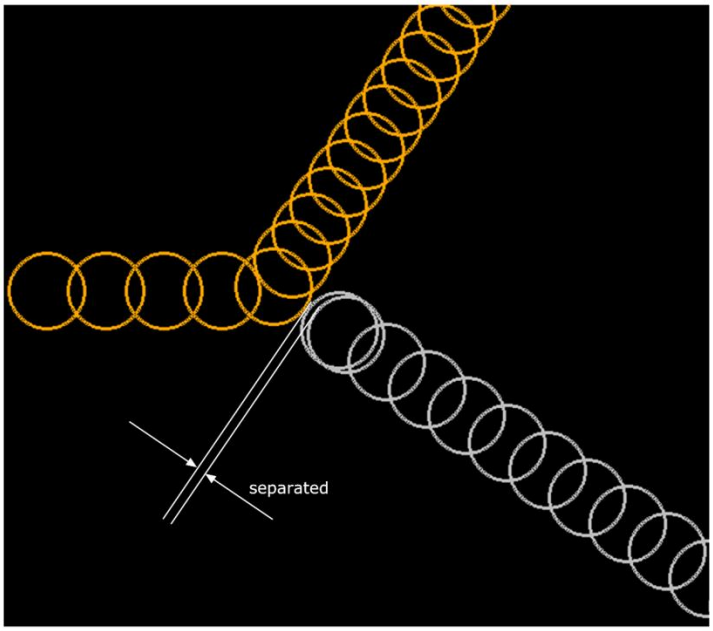


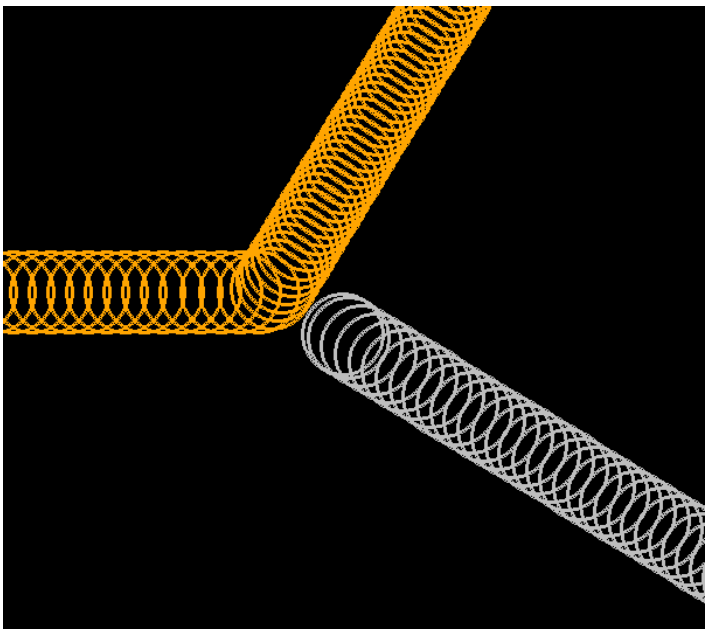
This correction puts the puck where it would have been if it had reversed its motion at the surface of the wall (no penetration). This correction can be done by moving the puck one penetration distance, x_p, out from the surface, along the direction of the normal. This is equivalent to splitting the correction into two parts: (1) reverse the incoming travel for a time equal to the penetration time duration t_p (penetration distance divided by relative speed of the two pucks along the direction of the normal). This is the time it takes to travel x_p at the incoming velocity. Then (2) proceed to move the puck, with the calculated post-collision velocity, for another time period, t_p. In other words, back it up in time to the contact point, then move forward to the time of the prior overlap (collision detection) using the calculated post-collision velocity.

For two colliding pucks, the two-step correction method is necessary. This is because the pre-collision and post-collision velocities differ in a way that depends on the mass of the pucks. While the end result of this correction will leave the two pucks separated by a distance equal to x_p along the contact normal, the final corrected positions of the two pucks can only be determined by backing up a time t_p to the contact point along the normal, then moving forward a time t_p with the post-collision-normal velocities. This correction along the normal is accurate if the collision normal establish during collision (overlap) detection is close to the normal at the perfect-kiss point (the onset of overlap). These two normals will be closer when frame rates are higher (shorter time step). This approximation is discussed in the *Final comments* section below.



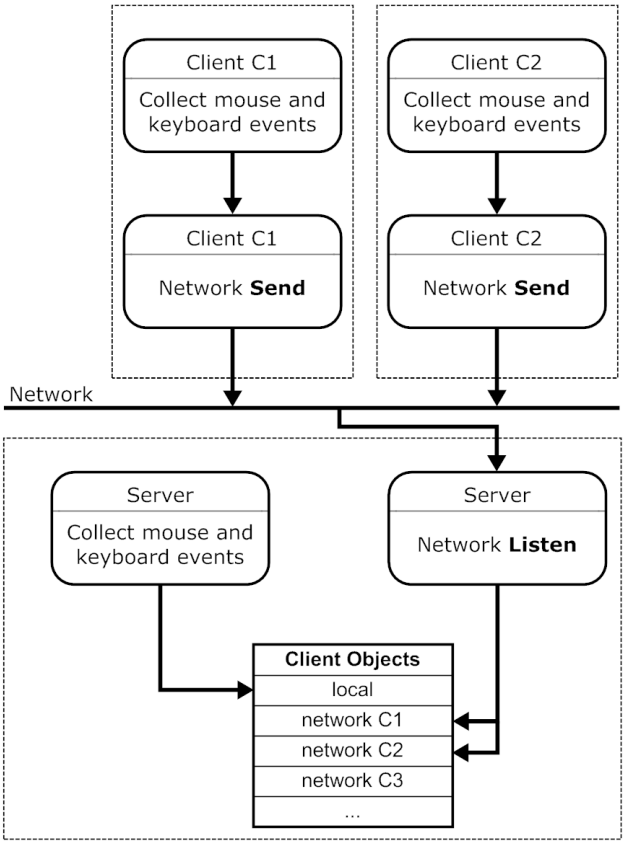
This is not a drawing but rather a screen capture of a slow frame-rate demo from an exercise script in this series. Erasing has been disabled, so a trail of images persists on the screen and can be recorded in a screen capture image. The incoming puck (orange) collides with a still puck (white). Penetration correction has been disabled. As is always the case with a still target, after the collision the incoming puck moves off in the tangent direction, and the still puck moves in the direction of the normal (similar to the drawing above). After the collision, the angle between the two puck paths is always 90°. The overlap which identifies the collision point is marked.



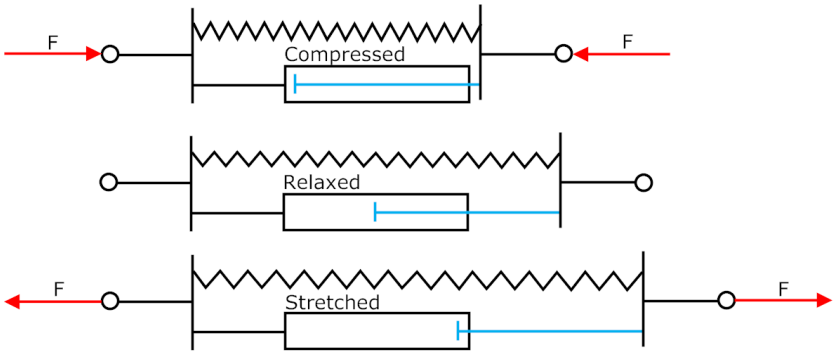


In the left image the penetration correction has been turned back on. The separation annotation indicates the corrected position of the two pucks after the collision. The separation correction is in the direction of the normal. The ratio of overlap to corrected separation will be, and should be, very close to 1.0000. (Again, refer to the *Final comments* section below for more discussion on the limitations of this approach.)

The right image shows the script running at a higher frame rate and with penetration corrections.



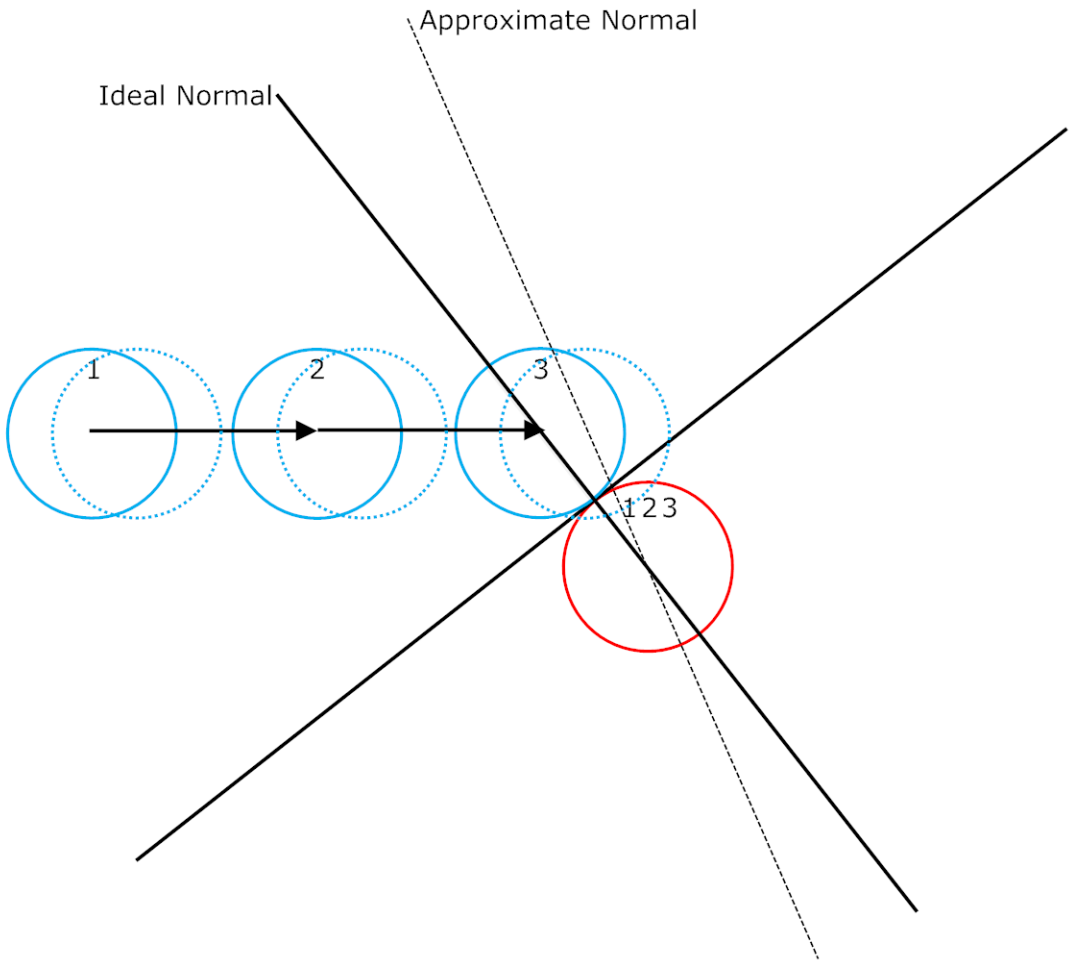
Implement the network client and server classes similar to what was done in the exercise on multiplayer games. But here the client data collected by the server will be stored in a list of client objects (as opposed to the dictionary data structure in the multiplayer exercise). The network listener identifies the client and then parses the data dictionary it receives into attributes of the client object. The server event queue is parsed and that data is put in the attributes of the local client.



Spring and damper concepts were used in the air-track exercises to model the effects of the cursor string forces. In this assignment some of the same ideas are used in developing the spring class. The drawing shows a relaxed system in the middle with no force applied to the two connected pucks. Two new things about this class: (1) a spring has a non-zero relaxed length (the cursor string is only relaxed if the cursor is over the center of a puck), and (2) the damper forces are calculated based on the relative velocity of the two pucks (cursor-string damping based on the speed of the selected puck). Note that spring and damper forces are not always in the same direction. For example, as a compressed system transitions to a relaxed state, the damping forces will oppose the spring forces.

Final comments:

- For anything but an absolute straight-on collision, the normal as established by the overlapping pucks is only an approximation of the (ideal) contact normal. The penetration changes the orientation of the contact normal. This is illustrated in the image below which shows both ideal and approximate-contact normals. The difference between the two is greater in high-speed-glancing collisions. When the single-step travel of a bullet puck is of the order of a puck diameter, the post-collision path as calculated with this approximation can strongly deviate from the true path that would be predicted from the ideal contact normal.
- If collision detection happens after the pucks pass the time where the two centers are the closest, then weird stuff will follow. In this collision case the pucks will be moving away from each other and the collision physics will basically reverse this and cause them to approach and possibly stick together. The normal will be strongly different from the ideal-contact normal. Again, the smaller the time step the less likely this case will occur.
- A more general (and perfectly accurate) approach is to back out the overlapping pucks to the ideal contact point and then determine the normal between the two pucks. This yields the ideal post-collisions paths and resolves both of the issues above. The details of this approach will be covered in the *Perfect Kiss* assignment (last one) before the Box2D section.



Python code: (see source code line on web page…)

If you can, try building this without major use of the posted source code. This is a big bruiser of an assignment…

Obfuscated code or incremental code didn't seem useful for this assignment, so no screen shots here.