Homework 1 Writeup

The method I used to get my character to traverse the edge of the screen was similar to a path following algorithm. I used an arrive steering behavior combined with a list of points to accomplish this. In effect, what happens is I have the character run arrive on one of the points until it reaches that point. Then I update the target point to the next one in the list. When it gets back to the starting point, it resets to the first target, causing the motion to repeat endlessly. The character's information updates based on real time, which is consistent between parts. I noticed an interesting interaction between the radius of deceleration, time to target velocity and max velocity while I was tweaking them. If the radius of deceleration is too small, relative to a given max velocity and time to current velocity, then the character will over shoot the target. This behavior could be useful for simulating situations where a character has a hard time adjusting its velocity, such as the ground being slick or the character having a large amount of momentum to overcome due to mass. Alternatively, if the radius is too high, the character will grind to a near halt in front of the target. The character will still make its way to the target point but will do so at an incredibly slow pace. For this particular situation, I found that setting the radius of deceleration, so it has a six to five ratio with the result of dividing the max velocity by the time to target velocity produces the most desirable result. The character will overshoot the target points by a small amount like this, which is relatively desirable since it makes it more readily apparent that accelerations are being used to produce the motion.

Much of what I said about the previous section applies to section two. The main difference is instead of arriving at preset points, the character arrives at the location of the previous mouse click or its current position if the mouse hasn't been clicked yet. Getting the character to arrive at the clicked spot was trickier than I originally thought it would be. Getting believably close was easy. The problem was the character would appear to stop and then start inching into spot. The most noticeable version was the one where it turned around and started moving again. This seems to have been caused by a small radius of satisfaction and a slightly off ratio between max velocity, time to target velocity and radius of deceleration. The small radius of satisfaction created a small stopping window for velocity and acceleration. With max acceleration held constant at one third of the radius of deceleration, the ratio for max velocity multiplied by time to target velocity and radius of deceleration that produced the best result ended up being around three to four. If the max velocity portion was higher or the radius lower, the character would over shoot and have to turn around, producing one of the worse results. If they were reversed, then the character stops in front of the point and gradually inches to the point. At around a one to one and one to two ratio respectively, the over and undershoot aren't significant and might not be noticed if the character doesn't actually need to stop at the point.

I implemented two different approaches to the wander steering behavior. The first one was the one we discussed in class. Each time it is run, a new acceleration and orientation are generated by selecting a point on an imaginary circle projected some distance in front of the character and running arrive on that point. The other approach involved randomly selecting a new orientation based on the current one and an angular offset on both sides. It would then run arrive on a location a certain distance from it current position along its new orientation. This offset distance could be modified by or even randomized, but all that would change is the magnitude of the acceleration in the chosen direction. In my case, I chose to keep it constant for simplicity sake. I kept the character on screen by causing it to wrap around to the other side when it tries to leave the screen. I chose this method because the character's orientation would be maintained during the wrap. This way changes in orientation are solely due to the wandering algorithm and not the character bouncing off of walls. When compared side by side, both methods behave similarly in the sense that they are functional wandering algorithms. The main difference I noticed seems to be the sense of urgency conveyed by the characters. My second method results in a slower character that makes faster changes to its orientation, while the first one results in a faster character that's slower to change its facing. I recognize that this difference could be due to my particular choice of parameters, but I tried to keep similar parameters equal where I could tell. For instance, I kept the offset to the imaginary circle and the offset along the new orientation equal. I think my second method better captures the concept of wandering. When I think of someone wandering aimlessly, I usually think of someone who doesn't really know where they are going. They're usually moving relatively slowly since they don't have any drive to be anywhere in particular and they change the direction they're headed on a whim. To me this sounds more like my second method than my first.

My flocking algorithm involved a hybrid approach to combining separation, cohesion and velocity matching. When there are no imminent collisions, acceleration is calculated by blending together the cohesion and velocity matching components with at two to one ratio between them. Normally I would've summed their weights to one, but I felt it took too long for the followers to achieve a reasonable speed like that. Since avoiding collisions with others is key to a believable flock and it returns zero when there's no imminent collision, I decided to have it over write cohesion and velocity matching when it returns something with a magnitude that's approximately greater than zero. In order to get it to function properly. For starters, I had to change how the wanderer reacts to stage boundaries. As it turns out, having the traveler suddenly teleport to the other side of the screen disrupts the current flocking motion. I thought it looked weird, so I switched it to have the wanderer bounce off of the boundaries instead. I also had to produce a variation of seek/flee that was capable of accounting for future positions. What I did was create a modify version of the existing code that takes in both the target and the current unit's future positions as parameters and runs the flee calculations based on those. While I was trying to get the collision avoidance to work, I found out that the time and distance thresholds work together to affect how tightly the flock members cluster together. The higher they are, the more spread out the flock will tend to be. If the lowest of these numbers gets too low, the flock will gradually collapse in to occupy the space of a single member unless other collision detection methods are employed. It's also the reason why my flock is proactive at evading collisions. The collisions are detected early enough in advance that the flock members have maybe too much time to react and get out of the way. The behavior seems to handle small scale increases to flock size fairly well, however I haven't tried large scale increases. I suspect that spatial limitations would break the behavior just because there would be a point where there isn't enough space to move and react. I did note that increasing flock size reduced the effect of the wandering unit, but that's to be expected since the behavior relies so much on averages across the flock.