Home Work 1
Building Game AI
CSC 584

Rishabh Sinha (rsinha2)

Introduction

Movement Algorithms are interesting to Game AI. Most games involve various AI techniques related to movement. These AI techniques can contain but are not limited to kinematic as well as steering behaviors such as arrive, seek, align, collision avoidance, separation, wander etc. Most Games have movement in some basic form and to implement any movement the above mentioned basic techniques are used.

Each of the above techniques can be implemented as a kinematic behavior or a steering behavior, the difference being that when they are implemented as a kinematic behavior, there is no notion of acceleration both in terms of directional (angular Acceleration) and positional (regular Acceleration). Thus movements in case of Kinematics are abrupt in nature. This is because in case of kinematics, since there is a lack of concept of acceleration, movement is either non existent or at full velocity. The same thing is true in case of rotation, where in either an object is not rotating, or else rotating at full angular velocity. However in case of Steering Behaviors, things are smoothened out and depict reality in a much better way, as sudden jerks are replaced by smooth transitions.

Character

The initial trivial task is the creation of the game character, which would be moved around the screen. This character, is shaped in form of a circle with a triangle attached on its head. To create the character, The ellipse function provided in processing is used to initially draw the ellipse, and then a triangle is drawn using the triangle function in such a way that, the vertices's of the triangle fit smoothly on the the surface of square. Hence creating our game character, on which the following set of movement algorithms will be applied. Figure 1 in the appendix depicts the game character developed in processing.

Kinematic Motion

The basic requirement of this part of the assignment was to depict kinematic motion using the character developed in the previous part of the assignment. And make the character move across the corners of the screen, on a rectangular path. To achieve this, I initially implemented a basic algorithm, where initially the character was present at the **top left** corner of the screen and took turns around the screen in a clock wise manner, This was implemented using simple kinematic motion initially, where in the character would reach the corner of the screen and then take a sudden turn at each corner at continue further The point at which the character would take a turn was determined by if statements Which ensured that whenever the character reached in vicinity of a boundary, it would turn and move towards the next vertex. This was the very first basic implementation of kinematic motion.

However, the problem statement required the character to start from the bottom left corner, the initial implementation was very easily adapted from top corner to bottom corner, by simply translating the drawing canvas and accordingly updating all the boundary conditions.

The third milestone in the process of development of the kinematic motion was the addition of breadcrumbs. Breadcrumbs are similar to footprints which mark how the motion has taken place. The problem statement required breadcrumb to be implemented so that the part taken by the character can be depicted in a clear and easy manner. To draw breadcrumb, the implementation in this homework uses an **ArrayList** to store **PVectors**,

as positions where breadcrumb have to be displayed. The need of ArrayList arises because of the fact that since a background is present, on the canvas, an object that is drawn, appears, just for a single frame and then it disappears. To make the breadcrumb permanent we have to store it in a List or an Array such as an Array list and display it at each frame. To store the breadcrumb, we just store a **PVector** mark the position of the breadcrumb and where the crumbs can be drawn at each iteration using the ellipse function.
These positions are obtained by using the **frameCount** variable. The implementation displays a breadcrumb, every 20 frames. This number required to be optimized so that the crumbs neither appear to far, such that the path is not clearly delineated nor they are close enough that they fill the screen, making the screen appear messy. These breadcrumb also have to be translated in equal amount to that of the character's initial translation, such that it appears that the character is dropping of crumbs as he is moving along the path.
The Last thing, to learn from this is to display the breadcrumb before the the display function that displays the character, The last display call appears on the top layer of the canvas. However, this can vary according to the requirements of the project.

However, after implementing the arrive behavior, It was evident that the same, thing could be used to achieve the motion in the first part of the assignment in a much more realistic manner. Thus I changed the implementation, to that dependent on **Arrive behavior,** where in at each corner of the screen, an arrive call to the next corner of the screen is used. The if statements are created, in such a way, that they turn the object towards the next corner point of the path, before it has reached the target. This is to prevent the behavior where the character comes to a standstill before taking a turn. For example if the character is moving along the top boundary of the path and arrive target is given as (950,50), The character is made to turn when x>920 so that it doesn't come to a standstill before turning. This makes the behavior look a lot more real.
Figure 2, depicts the motion of the character

Arrive Steering Behavior

The Arrive Steering Behavior, is the the second task that has been implemented as a part of this homework. The Arrive steering behavior, depicts the motion of the object while approaching a target. Thus when we say Arrive(target). We expect the object to either stop at the target location or stop near the target location. There are two important parameters that are provided to arrive function. These are the radius of satisfaction (**ros**) and the radius of deceleration(**rod**). The values of these parameters are very important while implementing Arrive. As the radius of Satisfaction is that distance, at which it can be considered that the character has arrived at the target. The radius of satisfaction is usually a very small circle surrounding the target as is kept to around 3 to 5 pixels. The value **3 to 5** is relative to the size of the character, which in my case is 75 pixels in diameter.
Basically the radius of satisfaction must not be that large that it becomes evident that the character has not exactly reached the target. That is in the case of the character used in the implementation, such a value would be around 15.

The second parameter that needs to be considered is the radius of deceleration. This is that distance at which, the character starts decelerating. Large values of radius of deceleration means that the character with start decelerating that far earlier. For the given implementation, considering the size of the background as well as the size of the character, a value of **100** for **rod** is suitable. On running this with a lower value of rod, would make it appear that the character comes to a standstill, by pressing hard on the brakes. Such a value for rod is 25. On the other hand, if the value of rod is too large say 500. The character would very rarely reach the max speed and again might not look good.

The figures 3,4,5 depict 3 different scenarios of Arrive steering behaviors. Figure 3 depicts scenario of discrete clicks of the mouse. And the next click is done after the character comes to a standstill after the first. Here if we observe the breadcrumbs we can see that through the spacing of the breadcrumbs it is evident that the the character accelerates at the beginning and again decelerates to a stop as the crumbs appear a lot closer to each other in those scenarios.

In the figure 4, we can see how the arrive would behave in case of a persistent mouse press, on comparing this image to the first we can see that the turns as are a lot more smooth. This is because, while turning, even though the character slows down slightly, it still continues forward at a fair velocity. This can be seen be the spacing between the crumbs as well, as once again the crumbs are closer at the turns that on the straights.

The figure 5 depicts the boundary case, where in the mouse is dragged outside the screen. Here we can see that the character wraps around and appears of the other side.

Wander Steering Behavior

Wander as name suggests describes the behavior where in the character moves literally on wanders around on the screen. To implement wander behavior, I have made use of the algorithm described, in class, where in 2 random variables are used to describe changes in direction of velocity (rotation). These 2 random variables are bounded by an upper and lower limit. These values are kept at (-.1,.1) radians. Further to prevent the orientations from changing too often (preventing wobbly look), The draw function calls the wander() function every 10 frames. On doing it too often, It would appear wobbly, however if a higher frameCount is used, It leads to a behavior where it appears that the character is seeking distinct points instead of actually wandering around the screen.

Wander Behavior can be seen in the figure 6 below, Here we can see that the wander behavior is as expected. This is because, subtracting two random variables with a common mean leads to very small changes in output.

On the other hand we can also see that when, we just use a single random variable instead of using two random variables, we get a lot more frequent turns, this is because, the the random value is not toned down by subtracting two random variables from each other. Thus we can see in figure 7 that, the path traveled by the character is highly convoluted.

We can also see the Figure 8 for the behavior of wander, without the use of frameCount to prevent wobbly behavior. Here, even though we cant see it clearly, through the screen shot, The Character, wobbles a lot as it moves through the environment, which might not appear normal. These wobbles, occur as the value or orientation, changes every frame making it appear that the character wobbles. To prevent this wobbly behavior, we make use of frameCount, so as to make the motion appear smooth and apply wander on every tenth frame.

Thus the best implementation of Wander behavior, can be seen when we use binomial random variables to implement changes in orientation.

Flocking Behavior

Flocking behavior is obtained by the process of blending. Blending is the concept of mixing together multiple steering behaviors. In case of the Homework Implementation, I have made use of separation, alignment and cohesion as required.
Cohesion, ensures that all those, other characters within a given region, seek each other, thus they stick together to each other. This distance is called the radius of cohesion **(100)**.
The second behavior added to it is alignment is a velocity matching behavior, where in the velocity of one the center of mass of all object within a neighborhood distance is matched this value of **neighborhood distance is taken as 100**. This value is taken as same as that for cohesion, is because, basically the same neighborhood is being is used for both the calculations. The third element in this blending is the separation. Separation is basically collision avoidance, that is if the characters come too close to each other. We apply an acceleration in opposite direction to that of the the oncoming character, so as to prevent the characters from colliding.

Flocking has been depicted below in the following Figure 9. Here we can see that the group of 40 characters are flocking together, We can play with the value of weights for the different behaviors, to optimize them, We get the optimized values on using weight of 2.3 separation, 1.5 cohesion and 1.5 alignment. If we increase the value of cohesion to one which is greater that the given value it leads to odd behavior which is depicted below in figure 10. Here we can see that the characters are almost on top of each other, Further with a low relative value of cohesion, the flock just breaks apart.

To make flocking work ,certain tweaks have to be made, to the algorithms, That is We have to carry out, cohesion, separation and alignment at a large scale level, that at a level of Lists, since we have list of characters. Further the value of the neighborhood distance needs to be played around with as, too small, cohesion or alignment neighborhood means, that the flock wouldn't hold together strong enough and would disintegrate. This can be seen in figure 11.

To manage large flocks, once again we require, a large value of neighborhood of cohesion, and alignment, to prevent the flock from disintegrating.

When you have two wanderers, The the other flock members, tend to follow the closest one, thus splitting the flock into two separate flock. In such a case, the clock members, follow the closest wanderer that falls in its neighborhood of cohesion, Similarly on the other side of the flock a similar situation will occur for few other flock members hence splitting the flock into 2 flocks. Such a split is depicted below in figure 12.

References:

1. https://processing.org/examples/flocking.html
2. https://processing.org/tutorials/eclipse/

Appendix :

Figure 1

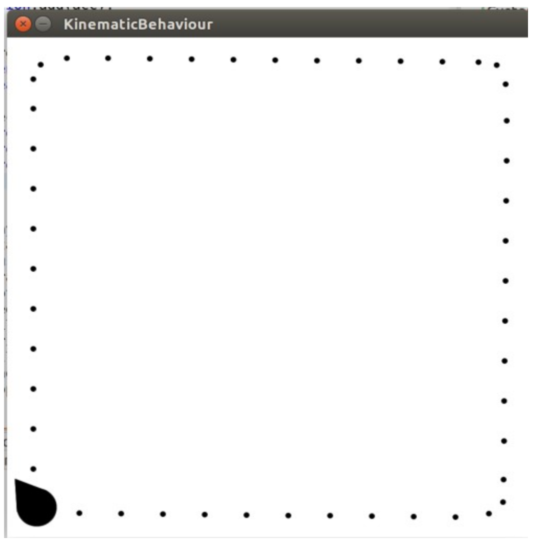

Figure 2

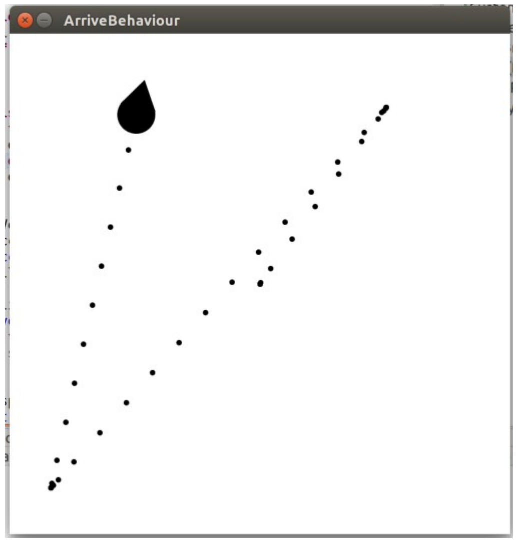

Figure 3

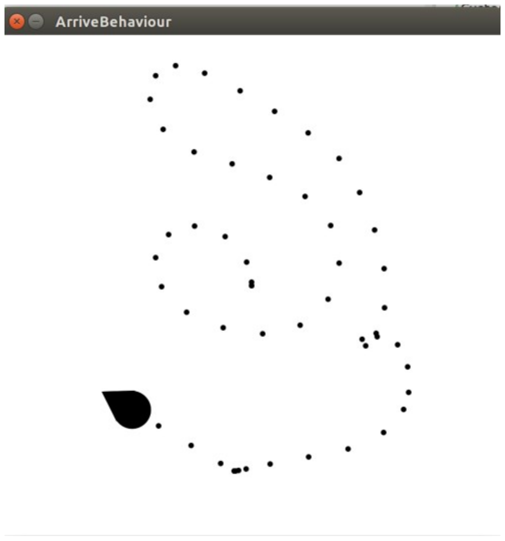

Figure 4

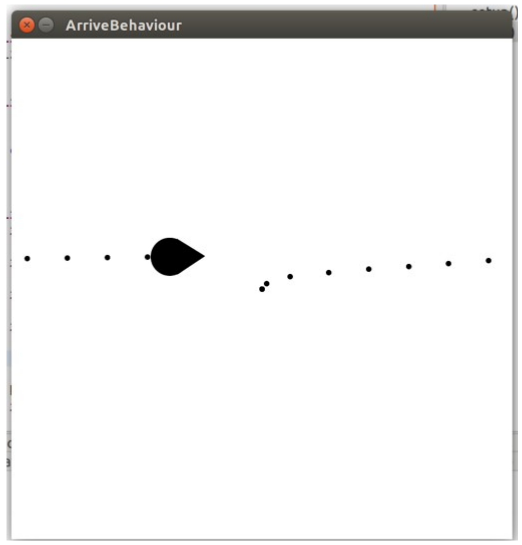

Figure 5

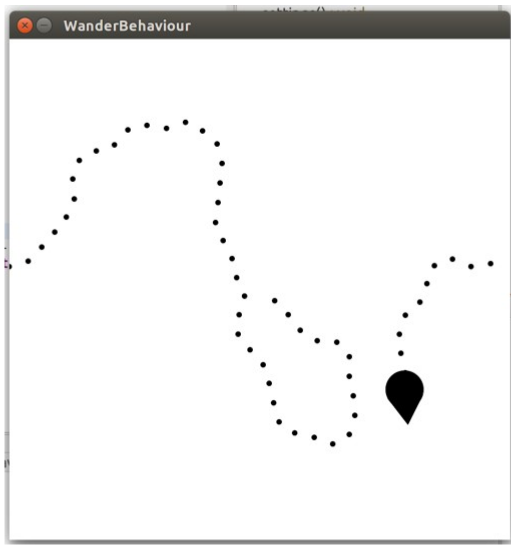

Figure 6

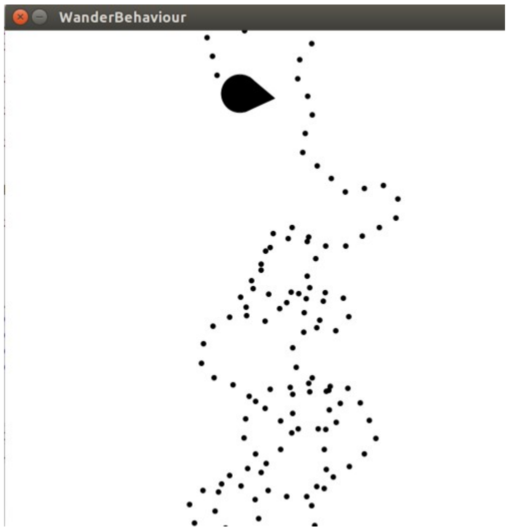

Figure 7

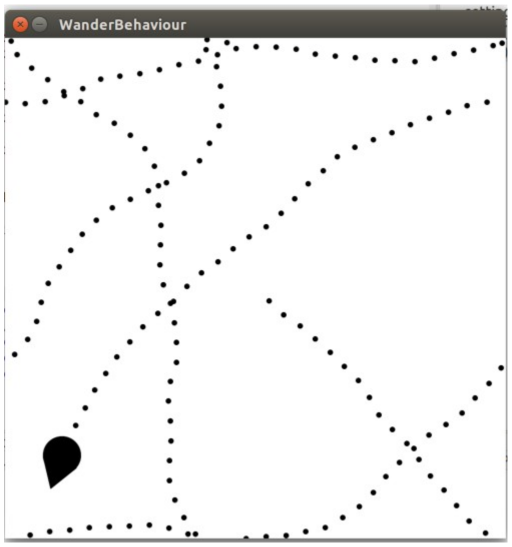

Figure 8

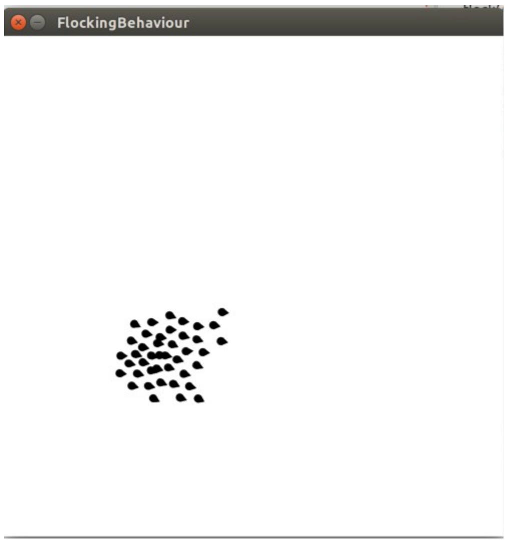

Figure 9

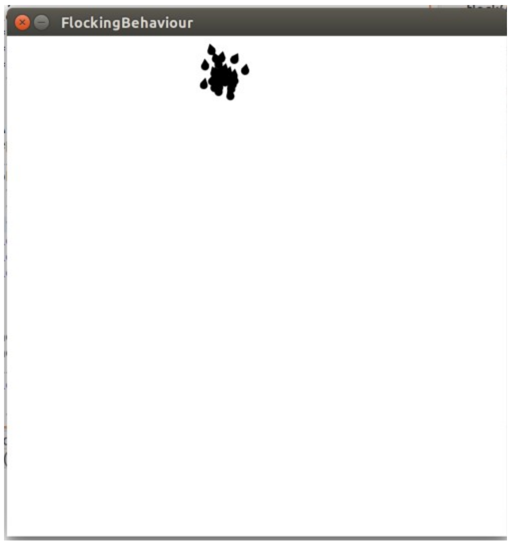

Figure 10

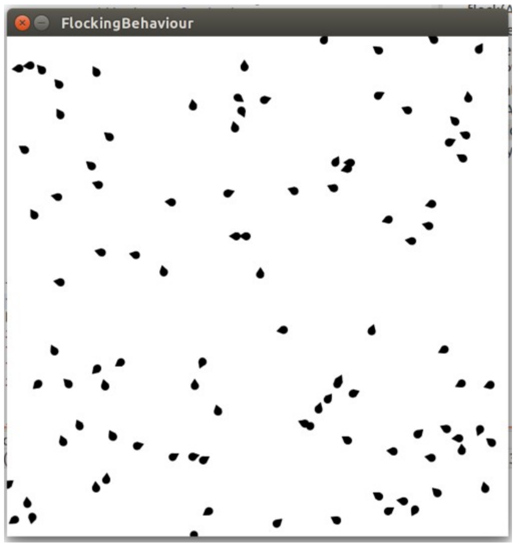

Figure 11

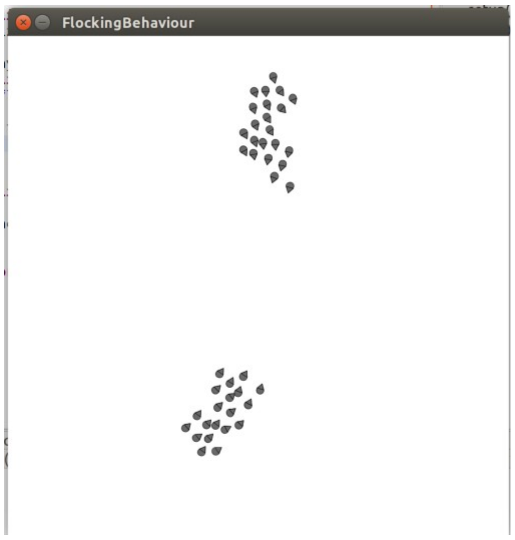

Figure 12