Dorian Rosen
Assignment 1

In this assignment, we started by building the framework for our movement algorithms and displaying them using the openFrameworks toolkit in C++. Part of why I chose to take this class was to continue mastering the C++ programming language, and to try to apply all of the things that I have learned over the last two years in my own personal work and through Joe Barn’s C++ game engine classes. In this write-up, I’ll break down the design choices I made, their implications, and some of the details of my implementation.

In class we defined a few data structures, the most important of which are the Kinematic, DynamicSteering, and something I like to call a Crumb. Each of the Boids that we’re drawing are represented by a Kinematic, whose associated movement data is updated by requesting DynamicSteering information from the various movement algorithms we’ll get to later. The implementation of Kinematic::ProcessSteering() is extremely important to making consistent behavior, since no matter how the steering algorithms calculate their output it should always be processed in the same way once it’s received by the Boid.

Now that these data structures have been detailed, we can start to talk about the steering behaviors that were implemented, and why there were implemented as they are. Deciding how to implement these behaviors was a really fun exercise, because there are so many different ways to do it. The design constraints that I had to think about were scalability, readability, and flexibility. The first option that I considered was creating different classes for each of the steering behaviors who all implement their own static versions of GetSteering(<Params>). This was appealing, because there was no need to initialize a “steering” object to access the various GetSteering() behaviors. This method also led to a straightforward function call. For example we may type Seek::GetSteering(<Params>). It is very clear which steering function we’re calling. One of the major downsides of this method, however, is that the parameters of the different same-class GetSteering() functions lead to really difficult to discern function calls. An example can be seen in the image below:

These GetSteering() functions are differentiated by a single additional parameter for the second call. Not very clear.

The next potential design pattern I considered was using a single MonolithSteering class that contains all of the different GetSteering() implementations for all of the various behaviors. This could be done with static methods or with an object of that class being instantiated. When I considered this option, I decided that the amalgamation of these methods made the most sense with each of the relevant parameters being set at construction of the MonolithSteering class. This would ideally reduce the potential for user error since all of the important variables are set only a single time. An added bonus of this pattern is that you could dynamically change a single variable like MaxAcceleration and you would see that impact every type of steering behavior in a consistent way. This greatly reduces the number of "magic numbers." Plus, tuning would be simpler (at the expense of precision). The major downside of this pattern is that each of the Boid's behaviors are going to be identical given the same steering function call. If you wanted to introduce randomness, or have any Boid individuality you'd have to design that into the Kinematic data type. It's also worth mentioning a design principle, the "Single Responsibility Principle." This advocates that a class should have a single responsibility, and any more than that can get difficult to manage and scale. I don't always agree with this principle, but in this case I feel strongly that it makes sense. This first assignment represents a boiler-plate setup we'll be expanding on throughout the semester, so keeping the project organized in a logical way is really important.

The final design pattern I considered is a combination of both of these previously described ones. In this third pattern a Steering object for each behavior is initialized with the necessary parameters provided, and GetSteering() calls of that type are called through their respective objects. This maintains the Single Responsibility Principle, and with a little care clears up the confusion shown in the first option. I think that this balanced option is ideal. I wish I had thought of it sooner! I plan to go refactor my code to match this pattern, but it'll have to wait for the next assignment.

Before we dive into the behaviors, we need some context of the simulation. First, if a Boid leaves the screen it is deleted. Second, the maximum speed of the Boids are controlled by the "Max Speed" slider which can be changed dynamically. Finally, the Crumb type represents the trail that Boids leave. They're initiated in an object pool, colored like their parent Boid, placed, and their radius scaled with delta time. Once they reach a certain size they're returned to the pool. Now on to the behaviors!

I started with implementing a Kinematic Seek. Unlike the Dynamic Steering algorithms, Kinematic Seek hard-sets the Boid's velocity and orientation, causing it to always face and move towards its target are the prescribed maximum speed. This algorithm is incredibly simple, and to be frank it's the least interesting in this assignment.

The next was Dynamic Seek, but before we dive into it, we should enumerate the differences between a Kinematic Steering function and a DynamicSteering output. As I mentioned, Kinematic

Dorian Rosen
Assignment 1

Steering methods change the active Boid's Kinematic properties, but Dynamic Steering methods do no such thing. Instead, they return a DynamicSteering output containing a linear acceleration and a new orientation target. These outputs are handled in a consistent way through the Kinematic::ProcessSteering method.

Back to Dynamic Seek. This algorithm has two 'modes.' The two modes are differentiated by their behavior approaching the target. The difference is shown below.

In both, the active Boid has an acceleration applied towards the target each frame. It's important to note that before accelerations are applied, the Boid experiences drag based on its Kinematic property. This way the Boid moves like a realistic object. Below are two gifs showing the difference drag makes.

As you can see, without drag the Boid moves like it's in a vacuum.

The next movement algorithm is Dynamic Wander. It works by accelerating the orientation of the active Boid in a direction and magnitude scaled with (random[0,1) – random[0,1)). This random expression gives a normal distribution centered at 0, with limits at +/- 1. This behavior looks a little bit twitchy, but behaves as you would expect. Below are two examples of Dynamic Wander, with different orientation variance values.

The final (and most interesting!) is Flocking. This works by linearly combining three different DynamicSteering outputs, scaled to produce the most appealing result. The three behaviors were: each flocking Boid must avoid all other members of the flock (Dynamic Flee), they must seek to match the velocity of the flock leader (Dynamic Velocity Match), and finally they must continuously seek the leader (Dynamic Seek). I noticed that to get the best behavior, the leader-seeking DyanmicSteering needed to be scaled up, and the avoidance scaled down. I also found that this simulation will continuously seek an equilibrium condition, and to avoid this it was best to introduce randomness. See snippet below.

Dorian Rosen
Assignment 1

This significantly improved the believability of the flocking behavior. Flocking can be set up to follow a wandering Boid, or the flock leader can dynamically seek the cursor. The latter is shown below.

There were a lot of really interesting elements of this assignment, and while it took a long time I really enjoyed doing it. I think that it’s really interesting that in a setup like this you can simply combine different steering behaviors to create emergent behavior. I can definitely see myself playing around with different combinations to see how they affect each other.