FIT3094 AI, ALife and Virtual Environments, by Alan Dorin.

Practical sheet: Vectors and flocking boids

Part A : Vector Class Examination

Locate the source code you downloaded for Assignment 1, *Pirates!* Open the source files SimpleVector.h and SimpleVector.cpp that were provided for you in the package. Study them to see how they work and answer the following questions.

1) Why are methods such as `output()` and `equalZero()` declared as `const`? What is the effect of declaring them in this way? Why do some methods have `const` references as parameters? What is the effect of declaring them in this way? What are the benefits?

2) Why is the value `mag` declared to be a private member of the class? What is an alternative way to code the method `magnitude( )`? What is the advantage of doing it the way it is currently implemented? When might this be a drawback?

3) Why is the value `v[]` declared to be a public member of the class? What is the advantage of doing it the way it is currently implemented? When might this be a drawback?

4) Why does the method `equalZero()` not just test whether or not `mag == 0.0`?

5) Copy this pair of class files for SimpleVector to a new directory for this lab. Extend the class in order to compute the dot product of two SimpleVectors. Do this by adding a new method `dotProduct()` (I.e. don't use an operator.).

Part B : Boids

Before you begin this exercise, you must have prepared by reading through the iBook, *Biological Bits*, sections 4.4 Particle Systems, 5.1 Physical Simulation and 6.1 Flocks, Herds and Schools. The purpose of this exercise is to build a simple two-boid flock using this background knowledge.

1. Design (*not* implement!) a Boid class that employs your SimpleVector for its calculations. The Boid class should contain at least a constructor, destructor and debug (`print()` or `toString()`) method and the following:

   - `canSee()` – return *true* if this boid can see a boid passed as a parameter
   - `tooClose()` - return *true* if this boid is too close to a boid passed as a parameter
   - `draw()` - draw this boid (see hint 3)

   - `calculateFlockCenteringVector()`
     Calculate a vector from the current boid's location to the position of a boid passed as a parameter if that boid is visible.

   - `calculateFleeVector()`
     Calculate a vector from the current boid's location away from the position of a boid passed as a parameter if that boid is too close.

   - `calculateAlignVector()`
     Calculate a vector in the direction of travel of the boid passed as a parameter if this boid is visible.

   - `updateVelocity()`
     Determine this boid's *future* velocity by: (i) calling each of the `calculateXXXX()` methods just listed to calculate flock centring, fleeing and alignment vectors; (ii) appropriately multiplying these vectors by scalars to use them as forces; (iii) computing an acceleration with Newton's Second Law of Motion for the current boid (Little hint:

each boid will need a mass data-member!) Lastly, using this acceleration compute a new velocity for the boid using Euler's integration technique.

- `updatePosition()`
  Update the *current* velocity for this boid from the *future* velocity, then move this boid by changing its position using Euler integration. (You can only do this safely after calling `updateVelocity()` for *all* boids - see Hint 2.)

Hints ————————-

1. Pass a parameter to methods `calculateXXXX()` that is a constant reference to another boid so that the boid doing the calculation can determine its local neighbourhood conditions using `canSee()` and `tooClose()`.

2. Since several boid class methods require you to change the velocity of a boid and the calculation of the alignment vector requires each boid to look at the velocity of its neighbours, it is important that you store the updated velocities and positions of the boids in temporary "future" velocity and position data members whilst you run through all the boids in the world. Then, when you have calculated new positions and velocities for *all* boids, you can set their "future" velocities and positions to be their "current" velocities and positions. This avoids allowing some boids to see into the future by mistakenly giving them access to the future state of their neighbours.

3. You might like to use the graphics code from your completed version of practical exercise 3 as the basis for drawing your boids.

Part C : Assignment Programming Commencement

Once you have finished your design for Part B above, implement and test your classes.

Declare two boids within the global scope and declare them `extern` within the source files that contain your GLUT eventhandler routines for: `glutIdleFunc()` and `glutDisplayFunc()`.
Within the `glutIdleFunc()` event-handler method update boid1 and boid2.
Within the `glutDisplayFunc()` event-handler method draw each of the boids by calling its class' draw method.