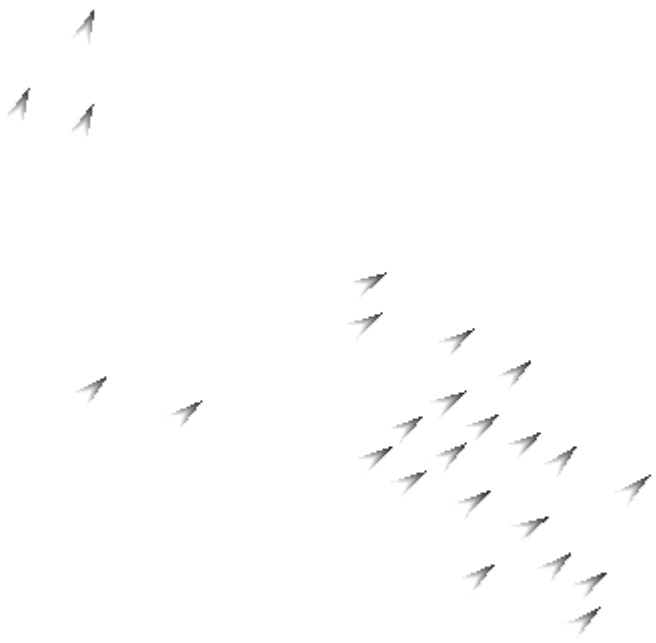
ocamlboids
visualizing a flocking algorithm

Serena Booth '16:
sbooth@college.harvard.edu
Jo Booth '15:
josephbooth@college.harvard.edu

May 5th, 2013



1 Overview

For our final project, we implemented a version of the "Boids" flocking algorithm, first published by Craig Reynolds in the 1987 ACM SIGGRAPH journal. We used object-oriented Ocaml and a set of openGL bindings to implement our version.

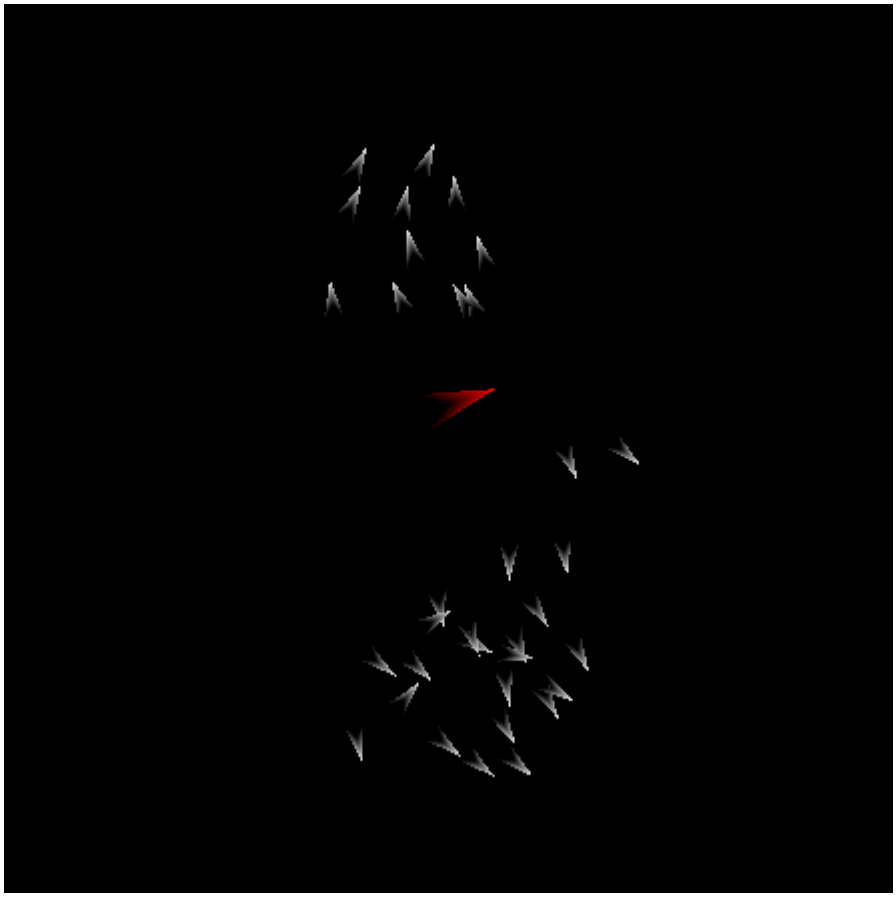


Figure 1: oh noes!
panicked boids fleeing the user-controlled "predator" (in red). This was one flock split into two sub-flocks by the fear algorithm.

2 Design & Implementation

Here's a thorough examination of our architecture and individual methods, covering everything related to the behind-the-scenes working of each of our creatures:

1. GLObjectI.ml

   GLObjectI is our most abstract class. This includes the framework for every functionality of members of GLObject. Namely, these objects all must have the following methods in common:

   - get_name
     Returns the name of each GLObject. Helpful for identification.
   - do_action
     Is used in GLObjects' initializers to call private methods
   - get_vel

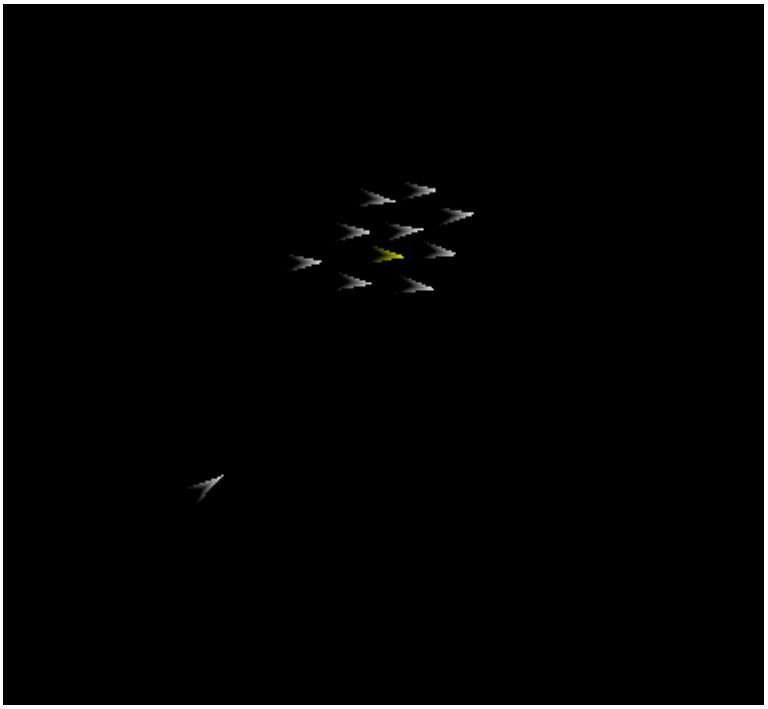


Figure 2: just trying to fit in.
The user-controlled "puppet boid" (in yellow) is "recognized" by other boids as one of their own. This feature was added to demonstrate the limited influence of an individual over the flock, and to allow the user to interact with the flocking algorithm directly.

Returns a tuple of velocity in the form (x,y), where x and y are floating point values.

- get_pos
  Returns a tuple of position in the form (x,y), where x and y are floating point values. Note that position and velocity are defined separately, but that velocity affects position.
- move_up, move_down, move_right, & move_left
  These methods are only implemented in the two objects which are user controlled.
- get_color_tuple
  Returns a tuple specifying R G and B values. This enables easy generation of custom actors (see "Predator.ml" and "PuppetBoid.ml").
- move
  Defines an object's movement, in conjunction with private helper methods.
- register_handler
  We credit pset 7 for register_handler. Ours works the same way.
- die
  Die allows us to know when to remove an object from our implementation of the world.

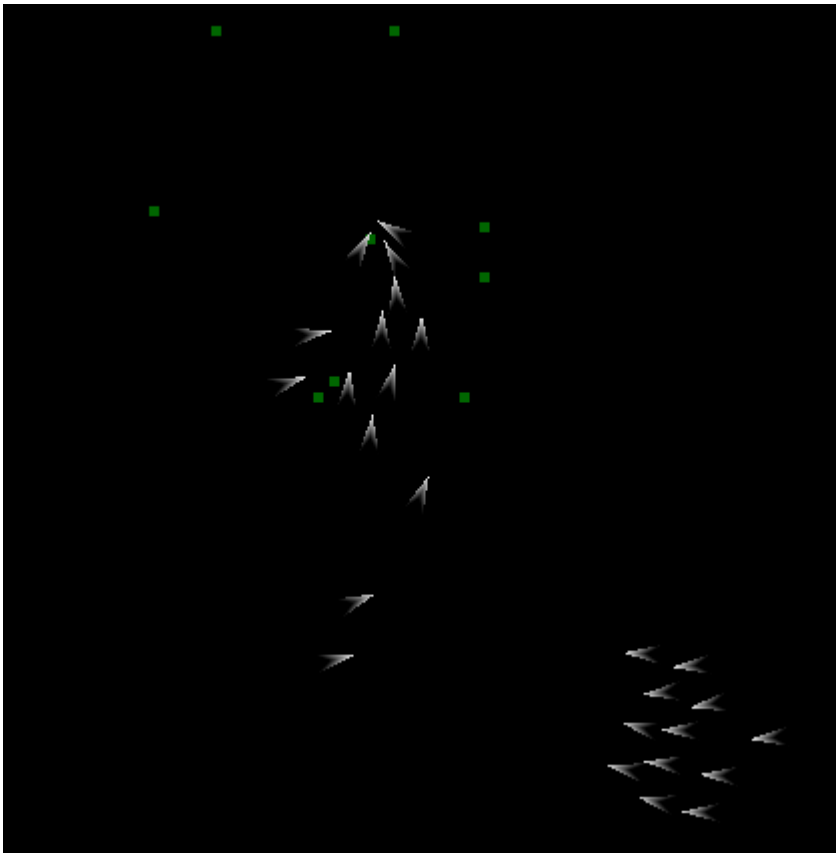


Figure 3: delicious green squares
A small flock of boids diverted by the presence of user-placed food particles (in green). This was included purely for fun; the boids' attraction to nearby food particles increases proportionally with $\frac{1}{r^2}$, where r is the distance from boid to food.

2. GLObject.ml

   In GLObject, we provide a general implementation for each of the methods specified in GLObjectI.ml. These are relatively straightforward, and generally describe a stationary, nondescript object.

3. Boid.ml

   Boid.ml implements GLObjects to fit the description provided in the 1987 ACM SIGGRAPH paper. These "boids" form the foundation of our project, and this is where most of our algorithms and complexities are introduced. We defer the discussion of these algorithms to the following subsection "Algorithms & Fun with Math."

4. Obstacle.ml

   Similar to boids, obstacles are implemented as children of GLObject. Obstacles repel boids. This is included in the "repel_rule" of the boid implementation.

5. Food.ml

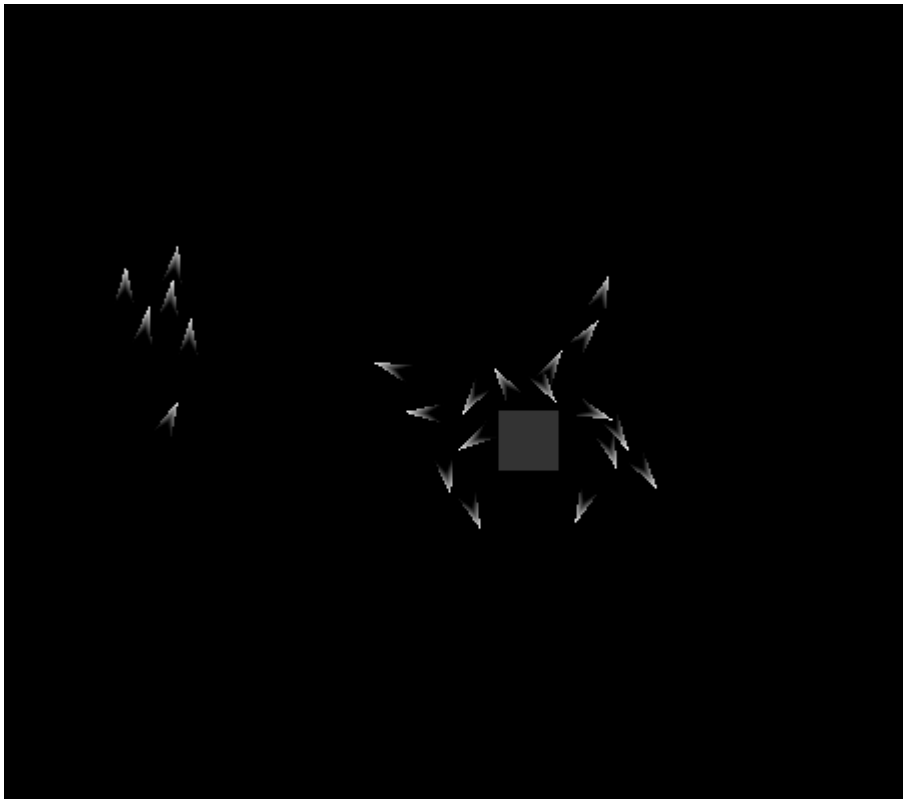


Figure 4: *collision course*
If flocks are given sufficient time to avoid user-placed "obstacles" (grey squares), they will subtly steer away from the obstacle to avoid a collision. When a block is placed directly in front of a fast-moving flock, however, pandemonium of this type ensues.

Food, again similar to boids and obstacles, are children of GLObject. Boids are attracted to food; this is included in the "repel_rule" of the boid implementation.

6. PuppetBoid.ml

   PuppetBoid's movement is controlled by the user through the WASD keys. However, PuppetBoid is recognized by other boids in the world as one of their own. PuppetBoid is essentially a way for users to interact with the flocking algorithms, to realize how much (or how little!) influence a single boid has over an entire flock.

7. Predator.ml

   Predator is extremely similar to PuppetBoid: it is again controlled through the WASD keys, but it serves a different purpose. The predator eats other boids, and it also envokes a fear algorithm which makes boids within a certain distance of the predator attempt to escape. The Predator class is extremely similar to PuppetBoid — the only difference between the two objects is their name, which enables different algorithms within the boid class to be activated.

We also include an implementation of:

1. World.ml

   The most important function of World.ml is to keep up with a list of all of the actors in a world. We do this by storing two lists: one which includes all the boids, food, and obstacles, and another list which keeps up with the puppetboids and predators.

   We use some interesting functions to work with the members of our World.ml lists:

   - World.fold

     If there is no predator or puppetboid in the world, World.fold invokes List.fold on the list of boids, food and obstacles in the world.

     If there is a predator or puppetboid in the world, World.fold uses the cons operator to include the predator or puppetboid as the first element of the world list.

2. Helpers.ml

   We include all of the functions we need to perform operations on vectors in Helpers.ml. These functions include vector addition, vector subtraction, vector scaling, vector normalizing, finding the angle describing a vector's direction, and finding the distance between two vectors.

3. glUI.ml

   - Our GUI draws upon basic linear algebra to render each boid as an appropriately scaled, translated, and rotated chevron, meanwhile food and obstactles are rendered as squares.
   - While the size for each object is defined in glUI, the color is extracted using the get_color_tuple method mentioned above. Its position is extracted from the get_pos method, and the heading is computed from its velocity vector (get_vel) using the heading method in Helpers.ml.
   - We bound the WASD keys to functions that adjust the velocities of user-controllable GLObjects. These objects are the predator and the puppetboid.
   - We bound other keys ('+') to spawn boids in the world and ('-') remove them from the world. We also bound keys to spawn the predator and the puppetboid. Another key ('R') is bound to allow users to remove obstacles from the world.
   - In addition to binding certain keys, we bound the left-click and right-click within glUI.ml. Left-click adds food to the world, and right-click adds an obstacle.
   - We render our boids using a gradient to give them softer and more natural appearances. Meanwhile, the movements we render are fluid and lifelike.

2.1 Algorithms & Fun with Math

2.1.1 The Core Algorithm

Boid behavior is governed as follows. Each boid runs the following computations on every "time event," thereby updating its position and velocity according to its environment. We

will define V_0 as the velocity of a boid at time $t = 0$, and P_0 as its position vector at this time. After a time event, a velocity adjustment vector V_{adjust} is added to the boid's velocity, and its "current velocity" $V = V_0 + V_{adjust}$. V is then tested to see if its magnitude is larger than an arbitrary "speed limit" l; if it is, we set $V := \frac{1}{|V|}V * l$.

Furthermore, after a time event, the position vector P is set to be $P_0 + cV_0$, where c is an arbitrary constant limiting the "weight" of the velocity vector. Note that since the GUI we are using has coordinate ranges $[0, 1.0]$ in x and $[0, 1.77]$ in y, each time P is set, the x component is computed *mod* 1.0 and the y component is computed *mod* 1.77. In practice, this has little effect on the behavior of the algorithm and allows the boids to "wrap around" in a sensible way.

We will now discuss the components of the adjustment vector V_{adjust}. Indeed, it is the generation of these adjustment vectors that arguably defines the basic behavior of the boids. As Craig Reynolds defines the algorithm, the boids are set to exhibit three behaviours: Separation, Alignment, and Cohesion. Mathematically, we will define each "behavior" as a velocity vector, and set

$$V_{adjust} = aV_{cohesion} + bV_{alignment} + cV_{separation}$$

where $a, b, c > 0 \in \mathbb{Q}$. Further, we will define $V(b_k)$ and $P(b_k)$ as the position and velocity vectors of the boid b_k, respectively.

Let B_{r_flock} be the set of boids b "near" our boid b_0 such that $|P_0 - P(b)| < r\_flock$. Then $B_{r\_flock} = \{b_0, b_1, ..., b_n\}$.

Now, conceptually, the vector $V_{cohesion}$ is simply the vector that begins at P_0 and extends to the "center of mass" of the boids in B_{r_flock}. It is defined as

$$V_{cohesion} = [\sum_{i=0}^{n} P(b_i) - P_0] * \frac{1}{n}$$

In practice, this vector serves to gather the boids close to one another.

$V_{alignment}$ is somewhat similarly defined, in that it serves to encourage boids in close proximity to mirror each others' velocities. This serves to make established flocks less visually "chaotic" in terms of their directional headings. We have

$$V_{alignment} = [\sum_{i=0}^{n} V(b_i)] * \frac{1}{n}$$

Finally, we have $V_{separation}$ defined as a "repulsion vector", driving a boid away from its closest neighbors. Thus we define a radius $r\_repel < r\_flock$, and a set $B_{r\_repel} = \{b_0, b_1, ..., b_{n'}\}$ where $|P_0 - P(b)| < r\_repel$ and naturally $n' \leq n$. Then we have

$$V_{separation} = [\sum_{i=0}^{n'} P_0 - P(b_i)] * \frac{1}{n'}$$

Note that, compared with $V_{cohesion}$, this vector difference is "reversed" - in some sense, while $V_{cohesion}$ functions as an "attraction vector," $V_{separation}$ therefore serves as a "repulsion vector."

Thus, if a boid's velocity at time $t = 0$ is Vo, then at $t = 1$, we have

$$V = V_0 + [a\sum_{i=0}^{n} P(b_i) - P_0 + b\sum_{i=0}^{n} V(b_i)] * \frac{1}{n} + [c\sum_{i=0}^{n'} P_0 - P(b_i)] * \frac{1}{n'}$$

And V is then normalized and scaled to the value of the speed limit l if its magnitude is $> l$.

2.1.2 Extending the Core Algorithm

In order to go beyond the boid behavior implemented by Reynolds in the 1980s, we added several additional features to the algorithms that define boid behavior. Thus the equation $V = V_0 + V_{adjust}$ does not tell the full mathematical story; rather, we add on a second adjustment vector V_{custom} that allows boids to interact with a) food b) predators and c) obstacles. Thus, in our implementation, we have

$$V = V_0 + V_{adjust} + V_{custom}$$

where

$$V_{custom} = V_{fear} + V_{food} + V_{avoidance}$$

.

Now, conceptually, the closer a boid is to a fearsome predator, the more rapidly it should maneuver to avoid that predator. Thus, in computing V_{fear}, if the position of the boid is P_0 and the position of the predator is $P(pred)$, assuming $d = |P_0 - P(pred)| < r_{predator\_sense}$ (that is, a sensory radius beneath which the boid may notice a predator approaching) we let

$$V_{fear} = \frac{c'}{d}[P_0 - P(pred)].$$

where d is defined as above and c' is an arbitrary constant. Intuitively, note that the magnitude of this repulsion vector is increases when the predator approaches our boid, as desired. Just for fun, we placed a further threshold on d such that if d is sufficiently small, the boid's "die" method will be called, signifying that the predator has eaten the boid. :)

While the predator's eating behavior is guided by user input, the boids' eating behavior is governed algorithmically. To simulate the behavior of a hungry creature, assuming $d' = |P_0 - P(food)| < r_{food\_sense}$ for some nearby food object (not necessarily the closest) we define the vector

$$V_{food} = \frac{c''}{d'^2}[P(food) - P_0]$$

where c" is an arbitrary weighting constant. Again, note that this vector is an "attraction vector," with reversed direction compared with V_{fear}. Further, the use of the inverse d'^2 multiplier results in a particularly dramatic "pulling" toward the food particle, which makes for a visually pleasing simulation of a hungry boid. As with the predator, if d' is below a very small threshold, the food object's "die" method is invoked and the food particle disappears, indicating that it has been "eaten" by the boid.

Lastly, our obstacle avoidance algorithm enables prevents birds from travelling within the user-placeable "obstacles," repelling birds that happen to contact the obstacle very

strongly (ie in proportion to inverse square distance). We let $r\_obst > 0 \in \mathbb{Q}$ and set $B_{r\_obst} = \{o_0, o_1, ..., o_n\}$ - in plain English, the set of obstacles within r_obst of our boid. Then we define

$$V_{avoidance} = \sum_{i=0}^{n} \frac{c'''}{|P_0 - P(o_i)|^2} * P_0 - P(o_i)$$

where c''' is an arbitrary constant.

Thus we have designed an additional adjustment vector which enables our boids to consume food, avoid obstacles, and run from predators; we have

$$V_{custom} = \frac{c'}{d}[P_0 - P(pred)] + \frac{c''}{d'^2}[P(food) - P_0] + \sum_{i=0}^{n} \frac{c'''}{|P_0 - P(o_i)|^2} * P_0 - P(o_i)$$

On a practical note, we rolled the computation of V_{custom} into the computation of the repulsion rule for concision. Therefore, the most accurate representation of V possible is in fact

$$V = V_0 + aV_{cohesion} + bV_{alignment} + c[V_{separation} + V_{custom}].$$

Not surprisingly, this algorithm generates behavior which is unstable and unpredictable. However, after any significant amount of runtime, boids are far more often found in flocks than on their own.

It's further worth noting that this algorithm's running time is $O(n^2)$, as each boid must gather information about every other boid in order to determine whether or not the other boid should influence its own movement.

3 Running the Project

Our code is dependent on the Lablgl package. This should be installed prior to attempting compilation. We have verified that this procedure works on both Windows and Macintosh computers. The user should ideally have "make" (from the freely available minGW) added to their PATH environment variable on Windows as well (on Mac, the same is accomplished by installing Xcode and the associated Unix command-line tools).

For your convenience we have bundled the latest version of Lablgl for Windows with the flocking program (see the "dependencies" folder). It is found in *lablgl-1.04-win32.zip*. We have also included the latest Lablgl source, although the authors of this project were not able to get it to build on Macintosh computers. Instead, we were able to install Lablgl and the included LablGLUT working on OSX 10.8 via the package manager Fink; once Fink is installed and configured to index unstable packages, running "fink install ocaml" and "fink install lablgl" will make compilation of the project possible.

On Windows machines, the contents of the Win32-lablgl archive should simply be extracted directly over the Ocaml installation directory - if Windows gives you requests to merge folders when you extract the zip file, congratulations, you got the target right (agree

to these merge requests). Then ensure that all of the folders in your Ocaml installation are in your PATH environment variable, and you should be off to the races.

To check, try running "lablglut planet.ml" from the command line in the /examples/LablGlut folder. If you see a planet, perfect; if not, your environment variables need to be adjusted or your installation of Lablgl was unsuccessful.

Once this is working, simply run "make" in the same directory as our project's files. It will generate "ocamlboids.exe", on which you should double click in order to see *boids*. In the unlikely case that "make" is unavailable, the sequence of calls to Ocamlc that eventually generate ocamlboids is included in "manual compilation instructions.txt" under /dependencies.

4 Operating the Simulation

Once the code is compiled, running ocamlboids should produce two windows - a small shell window listing the keyboard/mouse controls and their functions, and a large, blank Lablglut window. From here, the user can:

- Spawn boids at random by pressing or holding the '+' key, and remove them by pressing or holding the '-' key. The boids will be initialized at random positions with (small) random velocities, and they will rapidly gather together into coherent flocks.
- By left-clicking at any location in the World, the user can spawn a few "breadcrumbs" for the boids to eat. Note that the boids will be attracted to the breadcrumbs, but only when the breadcrumbs are within the boids' "sensing radius" for food.
- By right-clicking at any location in the World, the user can spawn an Obstacle at the location of the mouse pointer. These gray boxes will divert any boids who get sufficiently close, and boids will also avoid them if possible. Pressing or holding 'R' will remove these boxes.
- If the user would like to spawn a keyboard-controllable Predator, they can do so by pressing 'P', a toggle which spawns a stationary Predator (a large, red chevron) at a random location. Pressing 'P' again will remove the Predator from the world. This object is steered with the WASD keys, and if the user is able to bring the predator sufficiently close to a boid, the predator will then "eat" the boid. In practice, this is reasonably challenging to achieve. :)
- In order to interact more directly with the flocking algorithm, the user can also spawn a "puppet boid," by pressing 'H'. The puppet boid is identical in appearance to an ordinary boid except for its yellow color, and although it is user-controlled, the computer-controlled boids will interact with it as if it were a computer-controlled boid. Since only one movable object may exist in the world at once, if a Predator already exists, pressing 'H' will first remove the predator, while pressing it again will spawn the puppet boid. Pressing 'H' a third time will remove the puppet boid from the World. Like the Predator, the puppet boid is steered with the WASD keys. Note that the

puppet boid's maximum speed is slightly higher than that of ordinary boids, in order to facilitate the boid easily "catching up" with existing flocks.

... and they lived happily ever after.